MASTER OF PHILOSOPHY

The procedural generation of Vitruvian architecture

Noghani, Jeremy

Award date:
2015

Awarding institution:
Coventry University

The Procedural Generation of Vitruvian Architecture

By

Jeremy Noghani

April 2015

A thesis submitted in partial fulfilment of the University's requirements for the Degree of Master of Philosophy in the subject of Serious Games.

ABSTRACT

Can a single, cohesive framework be designed for the purpose of converting any set of historical, architectural descriptions into a digitally modelled format?

To address and exemplify this question, this study provides a set of novel and technical methods capable of converting a set of architectural writings – namely, Vitruvius' De Architectura – into an understandable set of procedural and grammatical rules that could then be incorporated into an application, providing a navigable digital model of a prototypical Roman city.

Different approaches were taken for the various elements of the generated city. A weighted formula was designed for the purpose of citing a city location upon a heightmap, incorporating factors like the distance to the nearest body of water and the gradient of the land. Three methods of situating generic structures within a city were proposed, including a probability distribution method that assigned buildings to districted allotments with a flexible degree of randomness.

For the generation of the building architecture, a novel formal grammar syntax was devised, capable of describing shapes in a deterministic and technical fashion. The grammar made use of superscripts preceding symbols for the purpose of notating conditional rules, and superscripts and subscripts following symbols for the purpose of adding attributes to said symbols. In this way, architecture was described using grammar rules in a way that would be impractical or outright impossible through the use of traditional grammar syntax.

The results produced by the application were deemed to be reasonably accurate. Slight discrepancies could be found between the architecture produced by the application and the architecture described by Vitruvius, but this was attributed more to the inaccuracies that arise from the transcription process than to errors caused by the grammars themselves.

The formal grammar syntax devised for the purpose of describing architecture was deemed to be effective for its purpose. Some cases of ambiguity and inconsistency were acknowledged, and suggestions were made for future improvement, but the syntax was largely considered to be suitable for its intended purpose of describing historical architecture in a technical and legible manner.

ACKNOWLEDGEMENTS

I would like to thank my partner and my family for providing continued emotional support and assistance throughout my entire academic journey at Coventry University.

I would also like to thank my work colleague, Dr Eike Anderson, for providing valuable input on the subject of historical accuracy, and for contributing his knowledge to the VAST 2012 journal paper.

Finally, I would like to thank my supervisor, Dr Fotis Liarokapis, for providing academic contributions towards the project at every stage, for offering constant encouragement, and for demonstrating unwavering patience and understanding.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF GRAMMAR DEFINITIONS

LIST OF TABLES

GLOSSARY

2D – Two Dimensional

3D – Three Dimensional

Araeostylos – A style of intercolumniation where columns are more than three diameters apart.

Basilica – A Roman public court building, usually positioned adjacent to the forum.

Cella – The enclosed inner chamber of a temple.

Centuriation – The Roman method of surveying the land and dividing it into a regular grid.

Cloister – An open space, surrounded by covered galleries.

Colonia – A designation for a Roman city of particular importance.

Colonnade – A row of columns.

Corinthian order – One of the classical orders of Roman columns. Notable for its ornate, leaf-adorned capital.

CR – City Radius.

De Architectura – "On Architecture" or "Ten Books on Architecture". This is the architectural manual written by Vitruvius.

DFC – Distance From city Centre. This is a measurement from a building's centrepoint to the designated city centrepoint.

Diastylos – A style of intercolumniation where columns are three diameters apart.

Doric order – One of the classical orders of Roman columns. Notable for its plain capital, and its relatively thick diameter.

Eustylos – A style of intercolumniation where columns are two-and-a-quarter diameters apart.

Forum – An open gathering space at the centre of a Roman city.

Insula – An apartment building or block.

Ionic order – One of the classical orders of Roman columns. Notable for the use of a capital that features a volute, a spiral pattern.

Orchestra – The low floor of a theatre or amphitheatre, usually containing seats for the senators and nobles.

Portico – The front porch of a building, usually featuring a colonnade.

Pycnostylos– A style of intercolumniation where columns are one-and-a-half diameters apart.

Systylos– A style of intercolumniation where columns are two diameters apart.

Triclinium – The formal dining room of a Roman house.

1 INTRODUCTION

1.1 INTRODUCTION

Procedural generation, the algorithmic generation of digital content, is a subject of increasing importance in the field of computer graphics. When a digital asset cannot be reasonably created manually, either due to the asset's complexity or due to a limiting time factor, then procedural techniques are employed.

Urban environments are one type of content where this is often the case. When a large number of varied building models are needed to fill a digital space, then shape grammars, split grammars, or Lindenmayer Systems may be employed to instantly fill the area with procedural content.

In the field of historical preservation, procedural techniques are often employed for the purpose of digitally recreating artefacts and architecture. Incomplete information retrieved from Photogrammetry and LiDAR data may require extrapolation via procedural techniques to form a complete digital model. Additionally, if there are not enough physical remains for a photogrammetric analysis, then models may be created based upon the writings of modern archaeologists, or historians of the era.

There is therefore a need for a defined formal grammar that can be utilised for the purpose of converting the guidelines and measurements laid out by historians into a standardised, mathematical notation, which can then be readily modelled and rendered procedurally by a program. There have been previous attempts in the field of procedural generation to convert recorded historical data into shape grammars, most notably by Mueller et al. (Mueller, et al., 2006) and by Yong et al. (Yong, et al., 2012). However, the focus has typically remained on the underlying procedural techniques themselves, rather than the formation and implementation of the shape grammars.

Therefore, the purpose of this thesis is to document the process of converting historical writing into descriptive formal grammars, and to then document the process of rendering the grammars as digital models within a computer application. The writings of Roman scholar Vitruvius are used to both test the system, and to illustrate its results. In particular, the architectural manual known as De Architectura, or Ten Books on Architecture, is the predominant focus of the study.

Throughout the thesis, we collectively refer to this assortment of grammars and methodologies as a framework. We aim to design the framework with De

Architectura specifically in mind, but also pay consideration to the broader possibilities of what may be required by a future user or developer of such a framework.

With this context in mind, we attempt to address an overarching question: can a single, cohesive framework be designed for the purpose of converting any set of historical, architectural descriptions into a digitally modelled format?

1.2 Aims and Objectives

The primary aim of this research is to provide a novel method of adapting technical descriptions of architecture into sets of grammatical rules and procedural techniques for the purpose of procedurally generating digital models. The methods by which these rule-sets can be implemented into a computer application, collectively labelled as our framework, are to be explained in detail.

The secondary aim of the study is to create a historically accurate digital representation of Vitruvian Greco-Roman architecture using the aforementioned grammar framework. The completed digital model can then be rendered, navigated, and measured in order to provide a testable analysis of the grammar rules' technical accuracy.

In order for these aims to be successfully fulfilled, a set of specific objectives must first be described.

The first objective is to provide a set of procedural methods, functions, or techniques that are capable of describing various details of an urban environment. These details include the location and size of a city, the road structure, and the location of various buildings within the city limits.

The second objective is to provide a method of defining the building architecture itself, in the form of a formal, context-free grammar. There are several criteria that the grammar must meet in order for the grammar to be considered suitable.

1. The grammar must be specific and precise.
2. The grammar must be flexible enough to encompass a wide variety of architecture.
3. The grammar must be free of ambiguity in order to make certain that the output results are deterministic.
4. The grammar ought to be comprehensible on a human level. This not only ensures that architectural rules can be properly written in the first place, but

also ensures that the rules can be read, understood, and adapted by others if needed.

The third objective is to use the constructed grammar framework to interpret the various architectural instructions written by Vitruvius, or other Roman historians and scholars where appropriate. Numerous pieces of Roman architecture must be converted into grammar rule-sets in order to create an abstract mapping of a prototypical Roman city.

The fourth objective is to describe the process of converting the procedural rules and grammar framework into a digitally modelled city. This creates two desired pieces of output: a digital city model file, and an encapsulating application that demonstrates the modelled city in a navigable and intuitive fashion.

The success of the system is tested by comparing the modelled, rendered results with technical drawings of Vitruvius' rules, and with real-world examples of the applied architecture. If the dimensions of the procedurally generated models and the historical examples can be said to be reasonably similar, then the implemented formal grammar can be described as historically accurate. If there are notable differences between the rendered results and the selected examples, then an analysis can be made of what caused the discrepancy.

Similarly, the chosen methods of siting a city and distributing the buildings can be tested by drawing comparisons between the output results and comparable historical measurements. In the instance of an inconsistency, a descriptive analysis can be made to assess exactly what caused the model to deviate from the historical record.

To gauge whether the overarching research question has successfully been answered, we must make an honest assessment of the extent to which the above aims and objectives have been fulfilled. If the aims have been met, as evidenced by a complete and cohesive framework, and by a functional model of a Vitruvian city, then it would be reasonable to suggest that the underlying framework has been successfully designed and implemented.

1.3 Assumptions and Limitations

It ought to be noted that the framework, the application, and the grammars described within this thesis are not intended to be all-encompassing solutions to the problems currently being faced in the field of procedural generation. To clarify this, and to properly contextualize the thesis, we have created a list of assumptions and limitations, detailing what lies inside and outside of the project's scope.

1. The subject matter of this thesis is architecture and elements of digital city models that directly pertain to architecture. Consequently, little attention is paid to the procedural generation of natural features, grass textures, and other superfluous elements of the landscape. Nonetheless, the procedural generation of these features is referenced under section 2.2, History of Procedural Generation, for the purpose of comprehensiveness.
2. We do not address the possibility of using intelligent agents to populate the digital city model. Although interesting in its own right, such an inclusion would not contribute to our outlined aims and objectives.
3. We examine previous contributions to the field of procedurally generated architecture in section 2.3, and provide a further critical analysis in section 2.4, culminating with a direct comparison in Table 1. We made the effort to be as thorough as possible, but we must acknowledge the possibility that a few notable contributors to the field have been unintentionally excluded.
4. Particular sections of Vitruvius' De Architectura are given limited attention, or are not addressed at all within the context of this thesis. These instances are denoted in their relevant sections throughout the thesis, but a further elaboration of the missed content can be found under section 5.3, Project Limitations.
5. The Implementation section of this thesis serves to document the process by which Vitruvius' rules are adapted into a digitally modelled format. Due to the nature of the research question, many subsections under this header can be read as both a methodology of our attempts to adapt a set of rules, and as a demonstration of the results. This is particularly the case with section 3.2.5, where defining the shape grammars of various structures functions as a demonstration of the shape grammar's success whilst still being just one stage of the architecture digitization process.

1.4 Breakdown of the Thesis

Chapter 2 illustrates the background, which has been split into several sub-sections. The first section, History of Procedural Generation, covers the broad scope of procedural techniques, from the early roots of Leibniz's fractal-like shapes, to the more modern applications. This paints a background of the context behind digital city generation, and gives an overview of some of the procedural techniques that are referenced throughout the project.

Section 2.3, Procedural Generation of Urban Environments, is a more in-depth examination of some of the techniques that have been used for the creation of digital

cities. Particular attention has been given to attempts to procedurally recreate historical architecture.

Section 2.4, the final part of the Literature Review, is a comparison of the techniques covered under the aforementioned Procedural Generation of Urban Environments subsection. The realism, efficiency, and limitations of each technique are discussed, and current gaps in the field are highlighted and evaluated.

Chapter 3, the Implementation section, is a detailed and technical explanation of how the procedural rules were devised, and how the proposed application was created. Due to the broadness of the application, it has been split into multiple subsections.

Section 3.1, Language and Graphics Library Decisions, explains the technical considerations that had to be made prior to the development of the application. Reasons are given for the choice of programming language and graphical library.

Section 3.2, Generation Engine, explains the methods used to adapt the writings of Vitruvius into procedural rules. Sections 3.2.1 through 3.2.4 systematically explain the procedural methods employed to adapt the various aspects of a Vitruvian city into digital form. Section 3.2.5 contains an account of the thought process behind the development of our proposed novel shape grammar syntax for the procedural generation of the historical architecture itself, followed by descriptions of our attempts to convert several archetypical Vitruvian structures into this syntax.

Section 3.3, Rendering Engine, describes the process of developing the application that encapsulated the procedural rules outlined in section 3.2. The steps taken to produce a DirectX model file from raw vertex data are described, and a brief explanation is given of the aesthetic and navigation development choices for the application itself.

Chapter 4, Results, demonstrates the output of the application. Sets of specialised renderings have been made in order to illustrate the various aspects of the project, and where possible technical comparisons are made to architectural drawings or real-life photographs.

Chapter 5, Conclusion and Evaluation, provides an overview of the observations, conclusions, and analyses that can be drawn from the data and rendered images in the Results section. We also offer a discussion of what has and has not been achieved within the confines of the project, and the potential for future work is discussed.

The final chapter, the Appendix, provides several pieces of code that we consider to be integral to the application. Section 7.1 contains CityMain.cpp, the “main” C++ code file that handled various aspects of the generation and rendering process. Section 7.2 contains BuildCity.cpp, the implemented form of the building shape

grammars described in section 3.2.5. Sections XFilePart1.txt7.3 and 7.4 are two key pieces of .X format code that are used in the creation of digital model files, as explained in section 3.3.1.

2 LITERATURE REVIEW

2.1 INTRODUCTION

As technology has evolved over the past century, there has been an increasing desire to push the boundaries of computer graphics, resulting in the production of increasingly complex digital models and renderings. The expansion of the special effects, film, videogame, and computer-aided design industries has contributed to a rising demand for a high level of realism in the field of computer graphics, especially with regard to three-dimensional (3D) digital models.

Traditionally, these digital models would be created manually, often programmed or sculpted on a polygon-by-polygon basis. However, as the complexity of models has increased, the scale of production grew proportionally. Consequently, the creation of high-quality digital assets now requires the skills of a team of modelling artists, texture artists, and animators. This is both costly and time-intensive, and as such there is a high barrier of entry for new companies attempting to break into any market that requires digital model assets.

Virtual buildings and cities are no exception to this problem. The creation of modelled cities has been an important and challenging aspect of creative industries. However, manual generation of large urban environments is time-consuming and tedious work, due to the repetitive yet varied nature of buildings and street layouts (Groenewegen, et al., 2009). As such, manually crafting a detailed replica of a city may require hundreds of man-hours of work.

One solution to this artistic and economic barrier is the use of procedural modelling techniques. In the past, procedural generation has been used for the purpose of automating the production of textures (Rhoades, et al., 1992), for the creation of trees and other self-similar structures (Oppenheimer, 1986), and for the creation of heightmaps that can be rendered as landscapes. More recently, combinations of procedural modelling techniques have been used for the purposes of creating populated, dynamic digital worlds. For these reasons, procedural techniques are often employed that make use of programmed rules to design aspects of cities automatically. The purpose of a procedurally generated environment is vital to how it is designed, and as a result the scope, detail, and interactivity of cities vary between projects.

2.2 History of Procedural Generation

2.2.1 Fractals

In the 17th century, mathematician Gottfried Leibniz documented his thoughts on self-similar and recursive shapes, which he labelled "fractional exponents" (Trochet, 2009) (Mandelbrot, 1982) . Two centuries later, other scholars like Karl Weierstrass and Georg Cantor extended Leibniz's work by producing branching and self-similar data sets that closely resemble what would later be classified as fractals (Trochet, 2009).

In 1904, dissatisfied with the abstract and "purely analytic" representations of fractals produced by his predecessors, Swedish mathematician Helge von Koch geometrically constructed a von Koch curve, and consequently the Koch snowflake (Koch, 1904) (Addison, 1997). The Koch curve was significant for demonstrating that non-tangential functions could be rendered using "elementary geometry", which helped bring together the geometric and analytic areas of mathematics that were traditionally kept separate (Trochet, 2009).

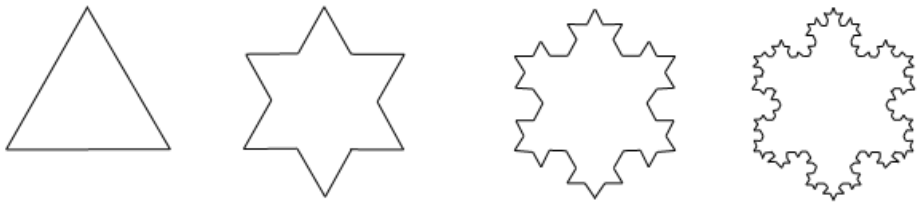

Figure 1: Progression of an iteratively-drawn Koch snowflake

In 1918, German mathematician Felix Hausdorff devised a key concept that extended the previous definition of dimensions in the context of topology and set theory, His work consequently led to the concept of fractal sets that exist in non-integer dimensions (Hausdorff, 1972)

At the same time Haudorff came up with his concept, two French mathematicians, Gaston Julia and Pierre Fatou, independently arrived at very similar results. They both studied the mapping of complex numbers and iterative functions, which led to the idea of attractors and repellors – points that attract and repel other points within a dynamical system (Lesmoir-Gordon, et al., 2000). Due to the limitations in technology, both Fatou and Julia could only produce what they could draw by hand.

In 1967, Benoit Mandelbrot wrote an essay, "How Long Is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension" (Mandelbrot, 1967). In this

paper he linked the ideas proposed by previous mathematicians with the measurements of coastlines, and proposed that the abstract concept of self-similar shapes and more observable natural phenomenon like coastline lengths were not so different.

However, it was not until later in the 20th century where the term 'fractal' was coined by Mandelbrot. Using computer simulations, Mandelbrot managed to render self-similar patterns in ways that were impossible for Fatou and Julia (Mandelbrot, 1982). By making use of the advances in graphics rendering technology, Mandelbrot was able to produce visual representations of sets, which helped form his Mandelbrot set and its accompanying fractal image. The fractal shape is created by parsing a complex number through a formula, and changing the rendering output if the result remains bounded .

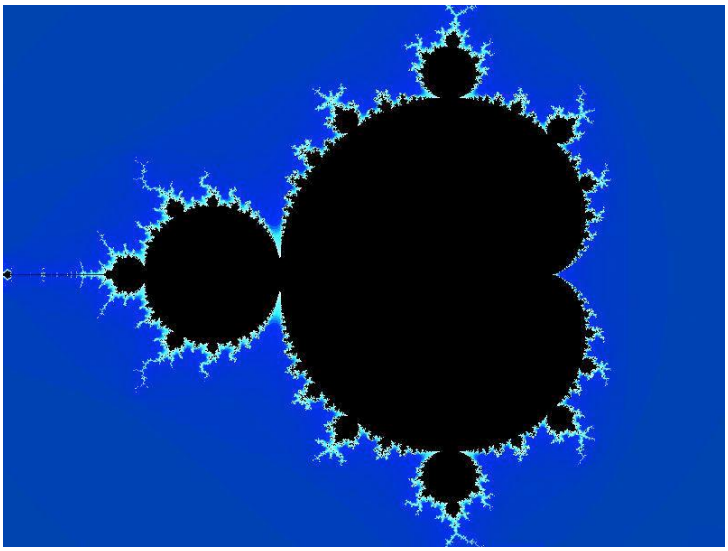

Figure 2: Example of a fractal representation of the Mandelbrot set. (Wikimedia Commons)

The Mandelbrot set is self-similar when magnified up to a certain point. Tan Lie proved it was "asymptotically similar to Julia sets near any point on its boundary" (Lesmoir-Gordon, et al., 2000).

Today, nearly all fractal studies are all computer-based due to the complex nature of the shapes involved (Pickover, 2009). The groundwork laid by fractal mathematicians has proved to be of utmost importance in the field of computer graphics and procedural generation.

2.2.2 L-SYSTEMS

In 1968, Biologist A. Lindenmayer devised the formal grammar known as L-systems for the purpose of documenting the growth and interaction of cellular organisms, such as bacteria and algae (Lindenmayer, 1968). The system functions by taking an initial object, known as an axiom (**ω**), and recursively rewriting variable elements (**V**) through the use of a set of rules, known as productions (**P**). This results in an increasingly complex hierarchical structure.

If certain elements or symbols of an L-system are represented with graphical substitutes, such as lines or polygons, then self-similar diagrams or models can be created. Due to this capability, L-systems have found frequent use in the field of computer graphics, where they are often employed for the purpose of generating self-similar natural features, such as trees, rivers, or terrain (Prusinkiewicz & LindenMayer, 1990). The hierarchal nature of L-systems makes them well suited to organic features, which appear to grow increasingly complex as the level of detail is increased (Lluch, et al., 2003).

L-systems have also been used for the creation of virtual roads (Parish & Muller, 2001). The self-similar, branching, interconnected nature of roads makes them comparable to plants (Lynch, 1960), and consequently a lot of the same production rules can be applied to both. However, aside from this exception, L-systems have found limited application in the creation of man-made structures (Mueller, et al., 2006).

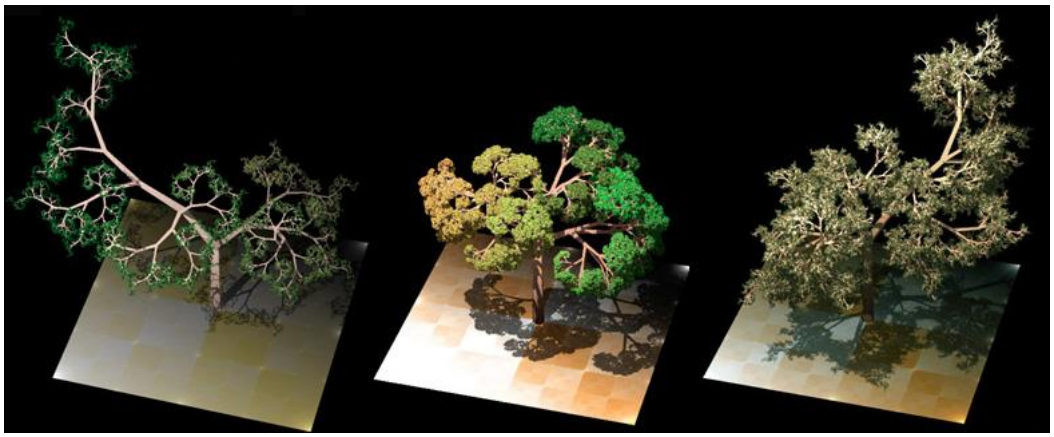

Figure 3: Three trees that have been produced using L-systems. A modified dragon curve has been used to define the spread of the branches. (Wikimedia Commons)

2.2.3 *PERLIN NOISE*

In 1982, Ken Perlin developed an algorithm capable of efficiently rendering semi-random textures with a seemingly natural appearance. The algorithm, which has since been named Perlin noise, was designed for the purpose of creating digital assets for the film Tron, but has since found use in a variety of graphics and modelling applications.

The algorithm works by generating a set of data points with randomly-assigned values based upon a seeded number. New data points are then created between the existing ones, and assigned interpolated values in order to create a smooth gradient. When these values are plotted upon a two-dimensional plane, a blotched noise effect is created. By repeating the algorithm at different levels of detail and merging the results, a fractal-like pattern emerges. By utilising different kinds of interpolation, and by placing weighted emphasis on different levels of detail, a range of effects can be created (Perlin, 1985).

Perlin noise has found practical applications in the field of procedural generation for the purpose of creating textures for clouds, landscapes, rocks and minerals, and other self-similar natural features. It is also often used in order to generate heightmaps for three-dimensional landscapes (Schpok, et al., 2003), and for the creation of disturbed water surfaces.

Comparable techniques have been employed for the generation of similar natural features. For example, midpoint displacement is an algorithm which alters the height of a point on a line in a hierarchal fashion, creating mountain-like silhouettes (Fournier, et al., 1982). When this is applied in two dimensions, such as through a diamond-square algorithm (Miller, 1986), an image or model is produced that is comparable to real-life terrain.

2.2.4 *Tile-based Systems*

In the commercial videogames industry, one of the most common methods of designing, storing, and rendering digital environments is through the use of tiles. A set of two-dimensional bitmap images is created by an artist. These images are then duplicated and positioned on a larger two-dimensional plane, and the position of each tile is recorded. In this way, a seemingly large bitmap can be generated from a handful of small images and a two-dimensional array of data.

Massively multiplayer online roleplaying games (MMORPGs) frequently make use of tile systems in conjunction with heightmap data and procedural techniques in order to efficiently produce varied large-scale environments for players to explore. By making use of probability distribution maps, the environments can be designed to ensure that the terrain is realistic, and that the maps are playable.

This image has been removed

Figure 4: On the left is an example of a set of tiles designed for a two-dimensional game. On the right is an example of a scene produced with the tiles. Note that the same tile can be used repeatedly in order to minimise memory usage. (Opengameart.org)

2.2.5 *Voxels*

Voxel structures are comparable to tile systems, in so much that they represent a digital environment by reducing a complex scene down to a repeatable, uniform grid. However where tile systems are traditionally represented as two-dimensional (2D) bitmaps, voxels are represented on a 3D, volumetric grid. Voxels are distinct from traditional 3D polygons in that the coordinates of each voxel node are not recorded individually, but are instead determined from the node's relative placement in the voxel data set.

Recently, voxels have been used for the purpose of rendering natural geography. The primary advantage of voxels over traditional 2D height maps is that voxels are capable of representing caves, overhanging structures, and other natural features that would otherwise be difficult to represent (Cui, et al., 2011).

In conjunction with image data, voxels have been used for the purpose of creating photorealistic reconstructions of scenery (Seitz & Dyer, 1997). By utilizing silhouette and stereoscopic data from multiple cameras, voxels can also be used for the purpose of recording and displaying human movements, as a form of motion-capture (Cheung, et al., 2000).

Voxels are actively used in the field of medicine for the purpose of quickly rendering three-dimensional sets of data. This includes the displaying of MRI data (Bullmore, et al., 1999)

Recently, voxels have been of increased interest in the field of commercial videogames, thanks in part to the success of the game Minecraft. The game made use of voxels in conjunction with procedural techniques to create large, navigable landscapes, often featuring natural geographical features and small urban structures.

2.3 Procedural Generation of Urban Environments

2.3.1 Photogrammetry

To gather information on the local architecture and street layout, various methods have been used in the past. LiDAR (Light Detection and Ranging) data is collected by using devices capable of sending and receiving laser pulses, to receive an approximate layout of an area in 3D. Laycock and Day, (Laycock & Day, 2003) outlined a method of using LiDAR data to quickly map a set of buildings. The authors also noted the flaws associated with this method, such as the inability to map the structure of a building's roof, but methods were proposed to automatically fill in the missing areas.

Photogrammetry, the method of retrieving geometric data from an image, is also used extensively. By utilising photographs taken at different angles, a textured virtual model can be drawn without the need for a traditional 3D artist (Kim, et al., 2000). This method has been explored and refined extensively, to the point that some researchers can successfully map a series of photographs of buildings to virtual models without the need for pre-planning (Pollefeys, et al., 2003).

R. G. Laycock et al. (Laycock, et al., 2008) later took these two methods a step further by focusing on reconstructing historical sites. The authors detail how they made use of LiDAR data and scanned town plans in order to automatically create digital footprints of building positions and shapes, which then had high-detail photographed textures applied. Where the data was missing, such as for roofs and damaged buildings, the areas could be filled in either through the use of procedural methods or through manual modelling. By repeating this process for both ancient and modern architecture in the same locale, researchers managed to create a 4D interactive program; the user could advance through time to watch how the architecture within a city had progressed.

The latest work in the field of historical recreation attempts to make the results as realistic as possible in order to make the user experience immersive. As a result, much of the latest research is in the department of accurate light scattering (Goncalves, et al., 2009) (Gutierrez, et al., 2008), and detail (Remondino, et al., 2009).

This image has been removed

Figure 5: An urban area of Kyoto, Japan that has been mapped from airborne LiDAR data into a digital model. (Susaki, 2013)

2.3.2 L-SYSTEMS

Muller and Parish (Parish & Muller, 2001) proposed a city generation program that made use of self-sensitive L-systems to automatically lay out a set of streets and generate virtual architecture. The encompassing application, CityEngine, was notable for making use of a procedural method that was typically reserved for natural phenomena, and successfully applying it to large-scale urban environments.

The application would start by taking a 2D height map, along with accompanying data on the desired city's vegetation, water boundaries, population density, road design, and so on. Road generation was then accomplished by creating branching paths that conformed to both "global goals" and "local constraints". Global goals would include steering the road system towards populous areas or away from undesirable terrain. Local constraints involved the checking of nearby roads and allotments to ensure that every new road section would be suitable within the confines of the application; roads could be snapped together to form junctions, if two road sections were to otherwise intersect or converge.

Buildings were generated by taking the allotments created during the road generation stage, and forming 3D architecture based on the underlying "footprint". L-systems could be recursively applied to the building geometry, causing the model to scale, translate, and subdivide, creating increasingly complex structures.

To add detail to building facades, each building face was split into a smaller grid, so that textures could be layered and applied according to what would be appropriate within each grid cell. The final system, CityEngine, was notable for being able to design large urban areas from scratch whilst still allowing for a high level of user customisation, including importing custom models and textures, or the writing of user-designed L-systems.

This image has been removed

Figure 6: A screenshot of the CityEngine software. (Wikipedia.org)

2.3.3 GEOMETRIC PRIMITIVES

Greuter et al. described a set of methods that allowed for the procedural generation of a 'pseudo-infinite' digital environment. The completed application, titled Undiscovered City, was capable of generating architecture on-the-fly, and of being navigated in real-time (Greuter, et al., 2003).

The application functioned by first laying out a uniform grid of roads, designed to be representative of dense, planned urban cities like New York. The square blocks between roads could then be allocated a hashed building seed, based upon the block's coordinates within the grid. The use of a hash function (i.e. taking the input of the grid coordinates and assigning a separate, fixed-length output) ensured that the assigned seeds would be random from the view of the user, but deterministic as long as the hash value is known.

With a building seed allocated, the geometry for each lot would then be generated. The application determined the height, width, and style of the building from the assigned seed. A floor plan would then be created by combining geometric primitives. By extruding this combined shape upwards, a three dimensional building section would be formed. A full building would then be generated by combining multiple building sections, with the higher levels being composed of fewer primitives than the lower levels.

To ensure that the application could be run in real-time, a specialised method of geometry culling was implemented. The technique, referred to as View Frustum Filling, functioned by only generating and rendering the buildings within the current cone of view. Memory was only allocated as needed, and consequently the application could efficiently and selectively render only what was needed of the large urban area. Additionally, a caching system was implemented to allow previously-generated geometry to be reallocated quickly, reducing the rendering times significantly under particular circumstances.

2.3.4 SHAPE AND SPLIT GRAMMARS

Alexander et al. (Alexander, et al., 1977) proposed a set of patterns that accurately described many types of structures, including buildings and road layouts. However, due to the non-formalised nature of these patterns, the language has proven difficult to transcribe directly into virtual environments.

George Stiny and James Gips devised a similar concept that they labelled as shape grammars (Stiny, 1975) (Stiny, 1980). That is, a set of formal production rules that specify how geometric shapes are created and transformed. The grammar is comparable to L-systems, in that the system iteratively replaces elements in order to form more complex and intricate patterns. However, whereas L-systems are expressed in a language of symbols or letters, shape grammars involve the direct manipulation of geometric primitives.

Shape grammars were successfully used in the analysis of modern and historical architecture (Flemming, 1987) (Downing & Flemming, 1981), and were later used to help define basic rules of how buildings could be generated in computer graphics.

Later, Wonka et al. (Wonka, et al., 2003) devised a variation on shape grammars for use in the construction of building facades, which they named split grammars. They proposed that, rather than assigning grammars on a per-object basis, a database of grammar rules ought to be used. This would result in an increase in the variety of

possible facades that can be created and a reduction in the amount of memory required, as well as allowing for the creation of large numbers of buildings in a relatively short time.

The split grammars themselves function by starting with an initial element, and then swapping shapes iteratively until a set of terminal shapes are displayed. The resulting set of shapes is then repeated horizontally and vertically within a set of parameters, creating an entire building façade. Contextual clues and control grammars are utilised in order to create variation in a logical, architecturally-sound manner, as well as to facilitate features like doorways.

More recently, shape grammars have been extended through the use of context-sensitive shape rules (Mueller, et al., 2006). Building on Wonka's previous work, Mueller et al. devised a system whereby 3D primitives are scaled and rotated semi-stochastically, and then assembled within a set bounded area. Occlusion query testing and "snap lines" are used in order to test for the intersection of the shapes, ensuring that building primitives fit together in a way that properly facilitates repeating façade textures.

Mueller et al. demonstrated the success of split grammars and context-sensitive shape rules with a procedurally-generated recreation of Pompeii (Muller, et al., 2005).

This image has been removed

Figure 7: A rendering of the virtual Pompeii model created through the use of split grammars (Muller, et al., 2005).

2.4 COMPARISON OF TECHNIQUES

It ought to be noted that the procedural methods employed for the purpose of city recreation and the methods employed for creating unique virtual cities from scratch should not be directly compared, but judged individually according to their purpose. City recreations are often created with historical accuracy as the explicit primary purpose, even if this requires time-consuming manual alterations or a lack of optimisation. By contrast, virtual architecture that is based on formal rules but not on any particular existing structure may be generated with aesthetics or user experience given precedence over architectural accuracy. Consequently, the criticisms and observations made in this section of the literature review are not intended to demonstrate one method's superiority or deprecation, but to highlight the relative differences between the demonstrated techniques.

2.4.1 REALISM

Laycock and Day, (Laycock & Day, 2003) noted that traditional methods of constructing 3D urban models from aerial and street-level photographs produced results that were fairly realistic, if impractical on large-scale projects. When 3D models created from LiDAR data are combined with textures gathered from accompanying photographs, the results are often considered to be close to photorealistic.

Muller and Parish's CityEngine makes effective use of L-Systems to generate a variety of street layouts and buildings within a set of constraints (Parish & Muller, 2001). The application has been effectively used to recreate a variety of real-life structures and urban environments, and in that respect the underlying techniques can be considered a realistic method of procedural city generation.

The grid network employed by Greuter et al. (Greuter, et al., 2003) for use in the Undiscovered City application cannot reasonably be considered realistic; the road structure has been described as artificial and overly homogenous (Kelly & McCabe, 2006). Additionally, the pseudo-infinite nature of the cities, although impressive from a technical standpoint, is not rooted in any real-world practicality. Nonetheless, the buildings generated by the application are both varied and aesthetically appealing.

Instant Architecture, the split grammar procedural city engine designed by Wonka et al. (Wonka, et al., 2003), appears to be capable of producing buildings of a fairly

realistic and varied nature. Some building facades appear to be unnaturally uniform, but more recent work with split grammars has attempted to address this issue.

2.4.2 *Efficiency*

Traditional photogrammetric techniques cannot be considered efficient under any capacity. Due to the specialised equipment and skills involved, the process of digitally capturing architecture is often restricted to those in the relevant academic fields. Additionally, the high level of detail associated with LiDAR scans often results in models being unsuitable for real-time applications until optimisations have been made.

Aside from the initial delay during the generation of a virtual city, Muller and Parish's CityEngine can be considered to be efficient. The application appears capable of generating cities on a large scale without difficulty, and appropriate level-of-detail checks are performed to ensure that the application remains useable even when a large number of assets are required to be displayed simultaneously.

The Undiscovered City application is notably efficient. Through the implementation of a culling technique known as View Frustum Filling, the authors have been able to ensure that the virtual environment is capable of being generated and navigated with relative speed on a range of delivery platforms (Greuter, et al., 2003).

Instant Architecture is reasonably efficient, and tests have shown that the application is capable of creating dozens of buildings that conform to the split grammar algorithms in seconds (Kelly & McCabe, 2006).

2.4.3 *Limitations*

Traditional methods of capturing model and texture data from LiDAR and photograph imagery is typically regarded as an arduous process that is difficult to apply to large-scale settings, such as for entire cities. Although recent efforts have been made to make aerial and street-level imagery more readily-available, such as through Google's satellite and Street View images, errors can be made when attempts are made to automatically map the data to a 3D model. The skill and time of an artist is therefore often required in order to ensure that the final model is accurate.

Both CityEngine and Instant Architecture suffer from the issue of repeating textures and models. Since the variety within the applications is dependent upon the underlying algorithms – L-systems and split grammars for CityEngine and Instant Architecture respectively – the generated cities are limited by the number and complexity of the input algorithms and source files. If a particularly varied city is required, then the work of an artist or seasoned program user is a necessity.

Similarly, the geometric primitive system underlying Undiscovered City is restricted by the number of textures and the complexity of the implemented shape-combining heuristic. Unlike CityEngine, Undiscovered City provides no immediately accessible method of adjusting the building heuristic, and so the application is limited in its versatility.

Generation technique	Advantages	Disadvantages
Photogrammetry	When effectively setup and applied, photogrammetry is perhaps one of the most accurate methods of mapping real-life architecture into a digital format.	Photogrammetric techniques alone are often not enough for generating cities when data is missing; other procedural techniques have to be employed. If no physical data is available at all, then the potential for photogrammetry is severely limited.
L-Systems	L-Systems are effective at producing dynamic, organic road structures and building layouts. The CityEngine application is a versatile tool with strong customisation capabilities.	Since each building type requires its own set of production rules, a varied CityEngine city can be time-consuming to create. L-Systems alone are of questionable use for the purpose of historical recreation.
Geometric Primitives	Buildings produced through the use of combined geometric primitives are visually interesting and varied. The Undiscovered City application is notably efficient, and is capable of producing cities of a particularly large size.	The buildings and roads that can be produced through this method are of questionable realism. The customisation options are restricted, and consequently the application has limited versatility.
Shape and split grammars	Modern split grammar techniques are capable of producing architecture that is realistic and varied. The techniques have been effectively used for the purpose of digitally recreating historic cities.	The realism and variety of buildings produced with shape and split grammars are dependent upon the complexity of the underlying production rules. Split grammar applications like Instant Architecture require a high level of expertise to be effectively used.

Table 1: A simplified comparison of the relative advantages and disadvantages of various city generation techniques.

2.4.4 *Our Work*

It would be simplistic to suggest that a single obvious gap remains unfilled in the field of urban procedural generation. Over recent years, numerous projects have been created to fulfil a variety of purposes, and many applications are malleable enough to suit needs outside of their original intended purpose.

Nonetheless, there is a particular niche that we believe to be underrepresented in the field: the adaptation of theoretical or written descriptions of historical architecture into digital model format. Vitruvius' De Architectura is one of the earliest and most notable examples of architecture in this written, descriptive format.

Photogrammetric techniques, perhaps the most widely used methods of adapting historical sites into digital geometry, remain unsuitable for the purpose of capturing written descriptions of architecture. Photogrammetry and LiDAR can only be effectively applied where some visual depiction of the architecture already exists. For a work like De Architectura, photogrammetry could only be used to capture comparative real-life examples of the described architecture. As demonstrated in Table 1, capturing the digital models of the descriptions themselves would be impossible through such methods.

The methods of combining geometric primitives, as exemplified in Greuter et al.'s Undiscovered City application, are also unsuitable for use in the adaption of architectural descriptions. These methods are well designed for the purpose of producing a large number of buildings with architectural variations, but they lack the precision and deterministic reliability necessary for the production of historical structures.

Techniques that create geometry from rules and grammars, such as Muller and Parish's CityEngine and Wonka et al.'s Instant Architecture, have been proven to be capable of describing historic architecture with some degree of success. It can be argued that the L-Systems and split grammars that underlie the respective applications are flexible enough to encompass a variety of scenarios, including the purpose of adapting written descriptions of architecture.

However, procedural rules and grammars typically remain confined to the context under which the original applications were made. If a particularly specific or customised scenario is desired, then a large amount of time would have to be spent deconstructing the application and repurposing it to suit the user's needs. It may be possible to construct a Vitruvian city using an application like CityEngine or Instant Architecture, but whether such an act would be practical or feasible with the tools presented by the authors is disputable.

We believe this to be an issue of specialisation. CityEngine has been designed to replicate sprawling, semi-stochastic, modern cityscapes, and the focus of Instant Architecture lies with producing variations from pre-established facades and architectural shapes. Neither was designed with the digital transcription of descriptive rules in mind.

Therefore, a relatively unexplored research gap exists: there is a lack of a framework that facilitates the simple adaption of historic writings into viewable, navigable digital models. Such a framework would be of particular use in the field of historic recreation for the purpose of creating digital replicas of historic structures that do not exist in any physical format. A Vitruvian city would be one such example of this; although Vitruvian-inspired cities exist in the real world, there are none that unwaveringly conform to every rule laid out by Vitruvius himself.

We do not propose to fill this research gap with an all-encompassing application, but we do aim to describe the implementation process for such a framework, to document and analyse the results produced by the project, and to note any issues encountered in the development process.

3 IMPLEMENTATION

3.1 LANGUAGE AND GRAPHICS LIBRARY DECISIONS

It was quickly decided in the planning process to use a low-level graphics library, such as OpenGL or Direct3D, rather than a higher level game engine, such as Unreal or Unity. This decision was based on the fact that the proposed application would require a large number of primitive shapes to be drawn, and so access to low-level drawing calls was deemed a necessity. Although a fully featured game engine lends itself well to rapid prototyping in many respects, the inclusion of extraneous features such as AI, animation, and audio support would only serve to bloat the application and hinder development.

A decision therefore had to be between the two most notable graphics libraries, OpenGL and Direct3D. Although other low-level graphics libraries exist, such as AMD's Mantle and 3dfx's Glide, OpenGL and Direct3D are considered to be the industry standard for use in games development and graphics rendering. Consequently, the libraries are very well documented, frequently updated, and highly capable.

OpenGL 2.1 was initially chosen for use in this project due to its cross-platform capabilities and the open nature of its license. However, problems were encountered during the programming of the application, and the switch was made to DirectX 9.0. DirectX does not contain any features that make it better suited to procedural generation, but in our scenario it did allow for a swifter development time, simply due to the nature of the documentation and the design of the rendering workflow.

C++ was chosen as the programming language to be used in the development of the engine. There were several reasons for this.

1. C++ is immediately compatible with both DirectX 9 and OpenGL; no time would be wasted on searching for and implementing wrapping libraries.
2. C++ is considered to be the industry standard for many comparable games and simulation applications. This ensures that the final application will be portable and sharable.
3. C++ is a versatile and powerful language that is well-suited to tasks like procedural generation and graphics rendering. Direct access to memory management ensures that RAM and CPU usage can be closely monitored.

3.2 GENERATION ENGINE

3.2.1 SITING SETTLEMENTS

The first asset to be generated in the project was the terrain that the urban structures could rest upon. The process of generating the terrain was straightforward, and comparable to the methods employed to generate terrain by Musgrave et al. (Musgrave, et al., 1989) or Ken Perlin himself (Perlin, 1985).

A 256 by 256 pixel Perlin noise image was generated in image editing software GIMP. The image was then used as a heightmap; rendered geometry was created by systematically drawing polygons for each pixel on the noise image, at heights corresponding to the darkness of the current pixel. A simple grass texture was assigned for the purpose of aesthetics and clarity, and an arbitrary waterline was assigned to create unbuildable and unnavigable areas of the heightmap.

The scale of the heightmap must be considered at this stage. For the purpose of our framework, we decided upon a scale of one pixel representing one hundred squared meters of tangible land. A single landscape would therefore measure 2560 meters on all four sides, and have a total area of 6.5536 square kilometres. Despite this, we still made calculations on a per-pixel basis. A single heightmap would have 65536 pixel-sized "points", each immediately adjacent to four "neighbouring points", that could be directly measured.

With these considerations made, a point must be chosen upon which to place the virtual settlement. A starting point could be decided by randomly selecting points until a free place is found, but this could result in cities positioned somewhat unrealistically, crudely, and possibly unbelievably in relation to the surrounding landscape. Instead, a more systematic approach must be taken. Three methods were proposed.

3.2.1.1 Method 1: Centremost Point Selection

The simplest solution would be to select the centremost point of the map. This would allow for a large potential city size, without fear of reaching the map's border. If the centremost point is unusable (i.e. the gradient of the terrain is too steep, or it is in a body of water), then a neighbouring point ought to be chosen. If this too is unavailable, then the check can be performed in an outwards spiral, until an

available point is found. There are many ways of performing this check, but a quick and crude method can be created through the use of two incrementing variables.

```
Check whether point (X midpoint, Y midpoint) is available
Else, until an available point is found:
{
     Increase Degrees by 1.
     Increase Distance by 1.

     Check whether point [Distance*sin(Degrees) ,
     Distance*cos(Degrees)] is available.
}
```

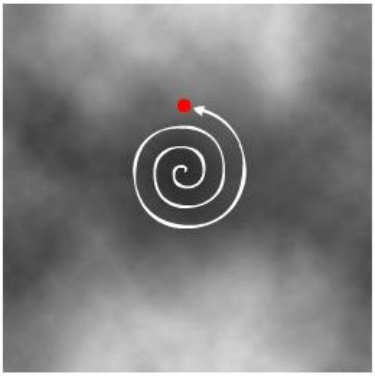

Figure 8: A mock-up example of the centremost point selection. The algorithm spirals outward until an available building point is found.

Although simple to implement, a clear issue with this method is that it is unrealistic. Assuming that the heightmap is randomly generated to some extent, it is unlikely that a potential Roman settler would choose the centremost point to start construction, as there is no geographical reason for choosing this location. This may be particularly noticeable if a city is generated far from a river, coast, or hilltop, as these are features that both Vitruvius and Roman settlers considered to be of great importance (Vitruvius I, 4)

3.2.1.2 Method 2: Factor Formula

An alternative proposed solution is a formula that can be called upon for each point of the map at the start of a program in order to determine where the "ideal point" for a city centre would be. Determining where an "ideal" position would be is somewhat

subjective, but by selecting key factors that are used in the decision-making process of siting an urban area in real life, we are able to make a fairly accurate assessment within the program.

There are a vast number of factors that are taken into account when choosing a site for an urban area, both in historical and modern contexts. However, it would be inefficient to incorporate factors that lie outside the scope of the generated heightmap. For example, Vitruvius recommended that a city should not be built downwind of a stagnant marsh (Vitruvius I, 4, 6), but no data on marshlands or wind direction exists in the heightmap. We did not consider the addition of such factors to be a practical use of time within the parameters of the project, and so these minor elements had to be excluded so that the more important factors could be implemented.

A key factor, both in modern and historical contexts, is the height of the land. Building at the lowest point of a valley is commonly seen as poor practise by urban planners, due to the increased vulnerability to natural disasters. Additionally, historical towns are frequently sited upon hills, as the height provides a natural advantage over potential invaders. The height of the current point being calculated (H) must therefore be incorporated into the formula.

As an extension of this, the gradient of the slope the point rests on (G) must be considered. If a point is particularly high, but rests on a steep slope (i.e. the point is on the tip of a mountain), then it is safe to assume that the land would be somewhat difficult to build on. For the sake of calculations, the gradient ought to be calculated as a scalar field, where 0 would denote a flat surface, and 1 would denote a 45 degree slope.

The distance to the nearest body of water (either a river, lake, or sea) is also a key factor (Vitruvius I, 7). In historical cities, having immediate access to fresh water from rivers would often be seen as a necessity, and having access to the coast would provide potential for trade. Access to fresh water is seen as less of a necessity in modern cities due to technological advances in water filtering and transport, but access to the coast is still seen as beneficial when possible, again due to the increased potential for foreign trade. Therefore, the distance to the nearest body of water (Wd) must also be incorporated into the formula.

Another vital factor is the distance to the nearest road. In both historical and modern contexts, it is unusual for a city to be created in an isolated spot, with no form of road access to nearby urban areas. It is far more likely for a new settlement to be positioned upon an existing road, as this provides easier access for construction and trade. Therefore, the distance to the nearest road (Rd) must also be considered. If no road exists on the map, then this value must default at 0.

The distance from the centre of the map (Cd) must also be considered. Unlike the other factors that were included for the sake of historical accuracy, this factor was added for purpose of keeping the sited location practical and sensible within the context of the application. A city sited on the edge of the heightmap may be aesthetically unpleasing, limited in its utility, or in a worst case scenario, unusable due to graphical glitches. By favouring central map locations, application-specific problems can be avoided.

One final factor that must be considered is the maximum height of the map (MH). The sole purpose of this factor is to give the point height (H) relativity. If a map is generated between 1000 and 1020 meters above sea level, for example, then the other factors would immediately become irrelevant due to the large numbers being used. However, by incorporating a relative measurement, the height factor can be scaled down to a practical level.

Using all these factors, a simple formula can be derived where the Point Value (PV) for every integer coordinate on the heightmap can be calculated. The point on the heightmap with the lowest Point Value can be viewed as an "ideal" place to site a settlement, and is flagged by the program.

```
PV = Cd + Wd + Rd + [G*(MH-H)]
```

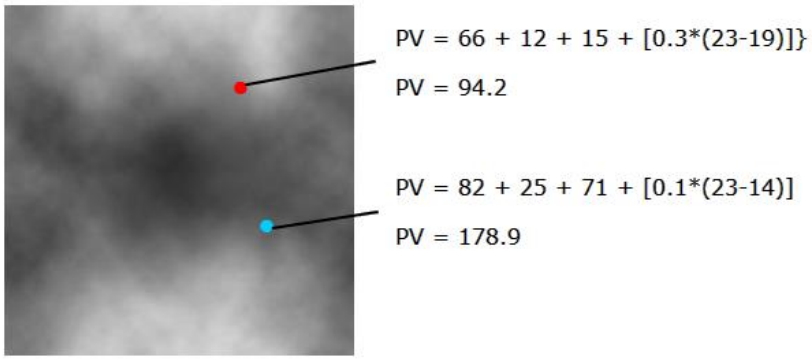

Figure 9: A mock-up example of the factor formula method. The red point denotes the point on the map with the lowest PV value. A cyan-blue point, denoting an arbitrary point, has also been included for the purpose of comparison.

Although potentially effective, a notable problem with this solution is that all of the factors are considered in equal measurements. This may cause issues in scenarios where a particular factor needs to be disregarded or given a disproportionate amount of precedence.

3.2.1.3 Method 3: Weighted Factors

To solve the problem of particular factors possessing too much or too little influence, a weighting system can be used. The simplest way of accomplishing this is to incorporate a set of "weight" variables that can be defined by the user, and that directly affect their corresponding factor.

For every point on the heightmap, a Point Value ought to be calculated, where distance from the centre (Cd), distance from the nearest body of water (Wd), distance from the nearest road (Rd), the gradient of the slope (G), height (H), and maximum height (MH) are calculated according to the heightmap, and where centre weight (Cw), water weight (Ww), road weight (Rw), gradient weight (Gw), and height weight (Hw) are defined by the user. As with Method 2, the PV with the lowest value would be the starting centre point of the city.

The final formula would therefore be:

$$PV = (Cd*Cw) + (Wd*Ww) + (Rd*Rw) + \{[Gw*G]*[Hw*(MH-H)]\}$$

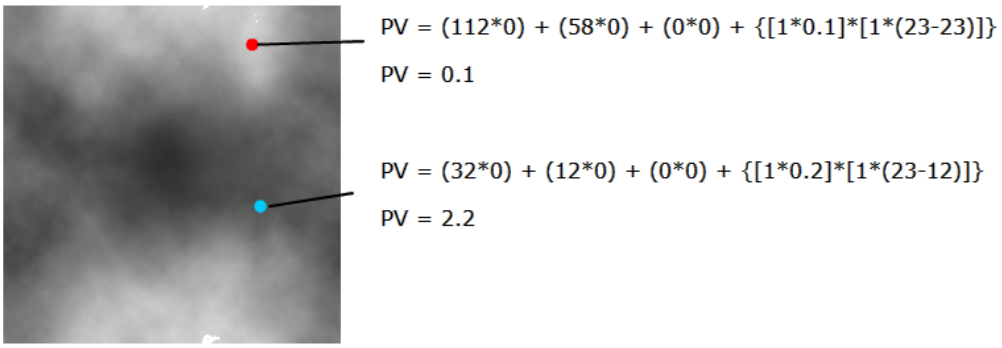

Figure 10: A mock-up example of the weighted factors method with a particular set of weightings applied. The red point denotes the point on the map with the lowest PV value. A cyan-blue point, denoting an arbitrary point, has also been included for the purpose of comparison.

The unit of measurement for the formula is irrelevant; as long as the same unit is used for Cd, Wd, Rd, Mh, and H, then it does not matter whether the chosen unit is meters, pixels, or an arbitrary unit used solely for the purpose of the program. Similarly, the weighting limit does not matter, as long as it is consistent within the formula. The reason for this is that the point with the lowest PV would be the same,

regardless of what units are used. If values wish to be compared across multiple applications, however, then unit consistency would be a requirement, but this is not necessary for our current needs.

On the off-chance that multiple points on the heightmap share the lowest PV, a simple test can be performed to see which point ought to be given priority. For the purposes of this application, in the case of a tied PV, the point with the lowest Cd value is selected as the starting point of the settlement. It should be noted that the chances of this occurring are particularly small, especially if the values are stored as floating points (as opposed to integers), and so this was not considered to be a major concern.

A major advantage of incorporating this weighting system is versatility. If the user wishes to build as close to the centremost point as possible (as in Method 1), then this can be calculated within the formula by placing a weighting value of 0 on every factor but Cw. This would result in the following result.

```
PV = (Cd*1) + (Wd*0) + (Rd*0) + {[0*G]*[0*(MH-H)]}
PV = Cd
```

Similarly, if the user desires a "realistic" application where the distance to the centremost point is ignored, then Cw can simply be set to 0, and the other weight values to 1.

```
PV = (Cd*0) + (Wd*1) + (Rd*1) + {[1*G]*[1*(MH-H)]}
PV = Wd + Rd + [G*(MH-H)]
```

As stated in section 3.2.1.2, certain elements that are not discernible from a heightmap alone have had to be ignored. Factors like vegetation, minerals, wind direction, and soil quality have not been taken into account, and this could potentially reduce the accuracy of the program if a truly realistic solution is desired. However, the ability to weight a set of factors ought to allow cities to be sited within a perceivable degree of realism, once the parameters have been adjusted to suit the scenario.

A final note ought to be made here with regards to the possibility of inadequate results being produced by unusual heightmaps or weighted values. For example, a heightmap with exceptionally high mountains towards the outer edge may result in a city being located at the very edge of the map, potentially causing structures within

the city walls to not be rendered. The limitations of our proposed formula are examined in greater detail and expanded upon in sections 4.1 and 5.1.1.

3.2.2 OUTER WALLS

Once the city centre has been decided, the next logical step is to mark the city limits and the buildable area that it encompasses. For a Roman city, this would frequently take the form of a city wall.

When taken literally, Vitruvius proposes that cities ought to be octagonal in shape (De Architectura I, 5-7). However, this was considered to be unsuitable for the application as "Vitruvius' scheme for an eight-sided town with a radiating plan was utopian and had no tangible impact on town planning practice in the Roman Empire" (Hebbert & Jankovic, 2009). Additionally, the use of an octagonal perimeter is not considered to be "central to Vitruvius' architectural theory" (Paden, 2001). Therefore, for the purposes of practicality, and in accordance to the historical record of many Roman cities, a 5-sided outer wall was chosen as the default shape of the surrounding perimeter.

Vitruvius did not specify the ideal size of a Roman city. However, from the archaeological record we can observe that the circumference of a newly founded city typically lay between two and four kilometres. From this, we can infer that the perimeter of the pentagonal wall ought to be between approximately 1.869 and 3.744 kilometres.

Vitruvius specified that the distance between the towers along a city's wall ought to be a maximum of one bowshot's length. In order to convert this measurement to a modern and implementable equivalent, the Roman military manual Epitoma Rei militaris was consulted. Author Vegetius stated that archers ought to train to fire their bows at 600 feet (Vegetius, ~390), or approximately 182 meters. However, this again deviates from the archaeological record, in which the towers on a city's wall were rarely placed more than 60 meters apart. It is possible that Vitruvius was not being entirely literal with his specified distance, or that he was taking the possibility of untrained bowmen into account when devising the measurement. Nonetheless, we chose to implement Vitruvius' measure for the purposes of consistency and aesthetic beauty.

Other specifications are also given. Vitruvius specified that the "thickness of the wall should … be such that armed men meeting on top of it may pass one another

without interference", and that "the towers themselves must be either round or polygonal" in shape (De Architectura I, 5).

With this information together, we are able mark out a precise pentagonal shape that encompasses the buildable interior of the city. The tower locations can then be flagged, and the structure is ready to be rendered when appropriate.

3.2.3 ROADS

With the city boundaries marked, the city interior can be generated. However, before the building structures can be designated and created, the city must be effectively divided into allotments or *insulae* through the creation of a cellular road system. We considered the possibility of generating the insulae first and designating roads in the surrounding regions, but this was considered inappropriate due to the lack of control that this order of generation offers over road parameters. Therefore, a road generation system had to be set up that first laid out the road structure, and then designated allotments to appropriate places on the roadside. Several existing methods were examined.

One method of generating roads that produces consistently natural-looking results is the use of L-systems, as initially described by Lindenmayer (Lindenmayer, 1968), and later effectively implemented into procedural city application CityEngine by Parish and Muller (Parish & Muller, 2001). L-systems are capable of creating branching, context-sensitive, and semi-stochastic lines (Stava, et al., 2010), which make them well-suited to the generation of roads with a rural or organic structural appearance.

An alternative type of road generation is the use of a grid structure, such as the one described in the Undiscovered City application (Greuter, et al., 2003). Although less flexible and less complex, this method of procedurally generating roads consistently produces straight, orthogonal pathways, effectively producing a gridded pattern. The Roman method of dividing a city into grid-structured, square allotments – a process known as centuriation – is a distinctive and influential feature of Roman cities (Romano, 2003). Due to the similarities between the method adopted by Greuter et al. and the pattern of centuriation found in the archeological record, a road system similar to the one described by Greuter et al. was adopted for the purposes of this project.

Much like the method used to determine the dimensions of the city walls (3.2.2 Outer Walls), Vitruvius' De architectura was used for the purpose of calculating the dimensions of the roads within the application. In the instances where there were inconsistencies or gaps in Vitruvius' description, the archaeological record was consulted.

Vitruvius did not specify the width of Roman streets. However, Roman law dating back to 100 B.C. specified that "Roman street width was fixed at a minimum of 4.5 meters [with] elevated sidewalks on both sides" (Mateo-Babiano & Ieda, 2005). From the archaeological record, we can determine that the width of significant

thoroughfares, including sidewalks, typically measured between 5 and 7 meters (Muller, et al., 2005). Vitruvius did not specify how large a city block or insula ought to be, but again the historical record can be consulted to gain a figure of approximately 73 meters squared.

Put together, this information allows for a detailed and historically accurate grid of Roman roads to be created, within the confines of the city walls. Sidewalks can be generated alongside each of the calculated roads, and the neighbouring empty allotments are ready to be assigned building structures.

3.2.4 BUILDING LOCATIONS

The next stage is to flag the location of important structures, starting with one of the centremost and most significant Roman structures: the forum. In towns located close to water with a harbour, Vitruvius suggests that the forum should be situated closer to that side of town, whereas otherwise the forum should be at the absolute centre of town (De Architectura I, 7). The settlement's communal areas with public buildings, including civic, religious and recreational buildings are located in close proximity to the forum (De Architectura I, 7; De Architectura V). In Roman cities the forum played a central role in the location of communal facilities, i.e. they were usually located at the forum itself as this created the focus for the settlement, or along the major axes of the town.

To apply this to the application, the grid square closest to the centre point is marked as the forum. The grid squares surrounding this are marked as the basilica, and local government. The grid square furthest from the centre point, but still within the bounds of the pentagon is marked as the theatre or amphitheatre.

The main sacred site of a city with temples for the highest deities and the protectors of the settlement should be at its highest point of elevation (De Architectura III and IV), easily reachable from the forum or the main roadways. In most settlements, the location of this temple district will be adjacent to the forum (in case the settlement is a colonia, this is referred to as capitol).

Vitruvius makes suggestions as to where the temples dedicated to different deities should be located. A temple to Mercury, god of trade and travel, should be in or alongside the forum of the settlement. A temple to Apollo, god of light and healing, should be adjacent to the theatre of the settlement if one exists. Temples to Venus, goddess of love, Ceres, goddess of agriculture, Vulcan, god of fire, and Mars, god of war, should be located outside of the city walls.

From this, we can infer that a temple ought to be situated immediately adjacent to the forum in our application. Additionally, temples may be flagged to appear on the outskirts, or completely outside the city walls, or on the highest heightmap point.

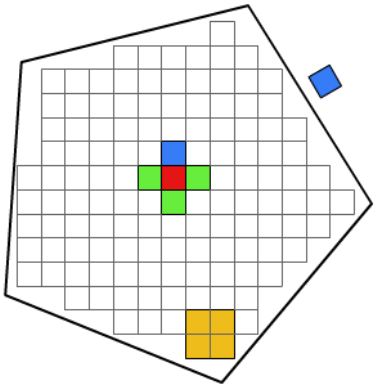

Figure 11: An example of the above Vitruvian rules applied to a city grid. In this instance, the red square denotes the forum at the centre of the city. The blue squares represent temples. In this case, there is one temple adjacent to the forum, and one outside the city walls. The green squares represent notable governmental buildings, such as the basilica. The yellow square on the city limits represents the theatre or amphitheatre.

The remaining squares on the grid can be considered to be generic or non-specialised structures. We have chosen to classify the generic structures into three groups, in a similar manner to the classification system outlined by Lechner et al. (Lechner, et al., 2004). The three building types are as follows.

1. Residential buildings. This would consist of one and two-story houses, terraces, and villas.
2. Governmental buildings. This would encompass those of a bureaucratic purpose, such as the basilica, as well as those designed to serve the public, such as the prison.
3. Religious and commercial structures. This includes smaller temples, shrines, shops, bathhouses, and other structures designed for the purposes of entertainment.

With the building types defined, a method must be devised for assigning these three types of structures to the appropriate grid square. Three methods were proposed and implemented.

3.2.4.1 Method 1: Random Selection

The simplest method of assignment would be to generate a random number for each remaining grid square, and to select a building type according to the number chosen. In this way, a fairly even distribution of the three building types will appear across the entire city area.

The method of generating random numbers ought to be addressed at this stage. For our application, a pseudo-random number derived from a linear congruential generator was used, with a seed value linked to time. By rights, this method should only produce pseudo-random numbers, not "true" random numbers, but in the context of our application we believe that this distinction to be irrelevant. Pseudo-randomness proved to be more than sufficient.

A basic pseudocode implementation of the first building assignment method can be seen below.

```
For each remaining free square:
     Generate random number R between 1 and 3
     If R == 1: Assign Residential
     If R == 2: Assign Religious
     If R == 3: Assign Governmental
```

The immediate problem that becomes apparent with this method is a lack of order, and consequently a lack of realism. Although it is possible for residential, religious, and governmental buildings to be built alongside each other, as is often the case with naturally developed settlements, such a layout would be unlikely in a pre-planned Vitruvian city.

3.2.4.2 Method 2: Sectioned Districts

An alternative method would be to section the city into districts. Dividing the total number of free squares by three results in a value indicative of the number of residential and religious structures needed to be built, denoted in the pseudocode below by the "NResidential" and "NReligious" variable counters. These counters are iteratively decremented and checked during the building designation section. The remaining space can then be attributed to commercial structures, resulting in three equally sized and homogenous districts.

```
NResidential = Total free squares / 3
NReligious = Total free squares / 3
```

```
Choose a random free square:
     Assign Residential
     NResidential = NResidential - 1
     Spiral outward and repeat until NResidential == 0

Choose a random free square:
     Assign Religious
     NReligious = NReligious - 1
Spiral outward and repeat until NReligious == 0

For each remaining free square:
     Assign Governmental
```

Although more realistic than the purely random method 1, method 2 suffers from being overly rigid. Without the use of 'fuzzing' – the use of randomness to create deviations in a pattern – the produced results may appear unnatural.

3.2.4.3 Method 3: Probability Distribution

A third method makes use of probability distribution in order to group together buildings in a semi-stochastic manner. One psuedocode implementation can be seen below, although it ought to be noted that there are other ways that this could have been implemented, such as through the use of Perlin noise. Variable "DFC" is used to refer to the distance of a building from the city centre. Variable "CityRadius" refers to the radius of a city at its widest point. As such, DFC divided by CityRadius would equal one at the outmost edge of the city, and would approach zero as we progress towards the center. Variables "ResP" and "GovP" are used to measure the probability of a residential or governmental building appearing on a particular grid square.

```
For each remaining free square:
     ResP = (DFC/CityRadius)^2
     GovP = ((CityRadius-DFC)/CityRadius-GovernmentalN

     Generate random number R between 0 and 1
     If R < GovP: Assign Governmental
     Else, if R > ResP: Assign Residential
     Else, Assign Religious
```

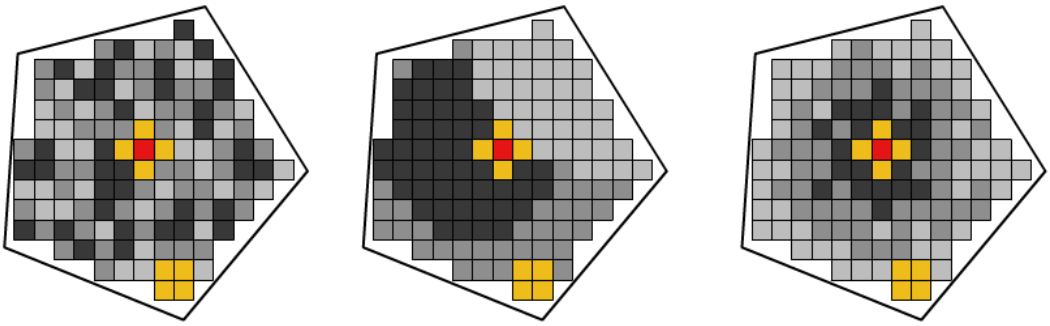

Figure 12: Examples of the three proposed methods of assigning building locations. Methods 1, 2, and 3 are on the left, centre, and right respectively. The red square marks the centremost point of the city (i.e. the forum), and the yellow squares indicate a special buildings, such as the coliseum or temple. The light, medium, and dark grey squares represent residential structures, religious structures, and governmental structures respectively.

3.2.5 Building Generation

3.2.5.1 Defining our Shape Grammar

At this stage, the application has a list of what building must be created for a series of grid squares upon a heightmap. However, no definitions have been made for how each building ought to be constructed. It would be trivial to design archetypal examples of such buildings in modelling software, and to simply import the objects into the application, but such an approach would not be procedural; since the explicit purpose of this application was to create a variety of structures on-the-fly, an alternative approach had to be taken.

It was quickly decided that a formal grammar ought to be used to define the overall architecture of each type of building, not unlike the shape and split grammars employed by Mueller, Pascal, Wonka, and other researchers in the field (Mueller, et al., 2006). When properly designed and implemented, formal grammars allow for clear, precise, and deterministic definitions of procedurally created shapes (Table 1).

In the language of a context-free formal grammar, a simple, multi-storied building with a roof could be described in the following manner:

Grammar Definition 1: Simple building

$$\begin{aligned} \text{Non} - \text{Terminals} &= \{B, F\} \\ \text{Terminals} &= \{W, R\} \\ \text{Start Symbol} &= B \end{aligned}$$

$$\begin{aligned} B &\rightarrow F, R \\ F &\rightarrow W, F \\ F &\rightarrow W \end{aligned}$$

Where **B** represents the entire building object, **F** represents a single floor of the building, **W** represents the four walls of a particular floor, and **R** represents the roof.

To break down the production rules in detail: our starting symbol, the building object **B**, is rewritten into the non-terminal floor object **F**, and the terminal roof **R**. The floor object is then rewritten into a set of walls and another floor, or into another set of walls with no additional floor. The final result would therefore be a building with at least one floor, each containing four walls, and a roof.

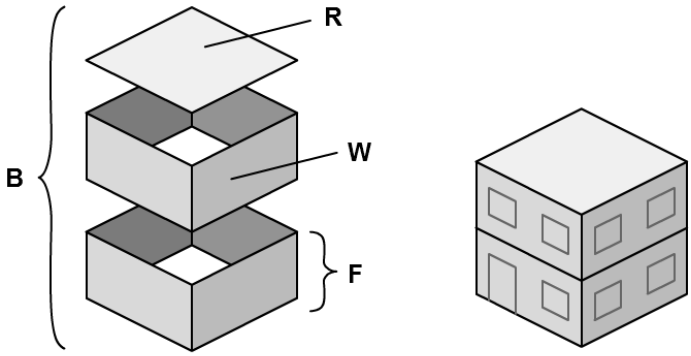

Figure 13: An exploded visual representation of Grammar Definition 1 (left), and a completed visual representation with an example façade (right).

However, there are several problems that immediately become apparent with a grammar this simple. The most immediate is that this is a non-deterministic representation of the building. By having two unclarified definitions of the production rule for symbol **F**, we are unable to say how many floors a completed building will contain; the production rule "$F \rightarrow W, F$" can run zero, one, two, or more times.

This problem can be addressed by adopting a deterministic context-free grammar. If, for example, it is known that the production rules were run in a strictly linear fashion, then we would be able to say with certainty that "$F \rightarrow W, F$" would run

exactly once, and "$F \rightarrow W$" would run immediately afterwards. The building would therefore have two floors, and there would be no room for ambiguity. However, simply adding a running order to the grammar would be a cumbersome way of defining more complex structures; the grammatical definition of a single building could potentially become hundreds of lines long.

An alternative solution would be to add comparisons to the production rules. A simple method of doing this has been outlined in Grammar Definition 2. The method of adding superscripts and subscripts to split grammars as described by Stephen Mann (Huang, et al., 2009) served as a foundation for our adopted syntax.

Grammar Definition 2: Simple building with conditions and attributes

$$\begin{aligned} \text{Non} - \text{Terminals} &= \{B, F\} \\ \text{Terminals} &= \{W, R\} \\ \text{Start Symbol} &= B \end{aligned}$$

$$\begin{aligned} \mathbf{B}^{\,flo} &\rightarrow \mathbf{F}^{\,flo}, \mathbf{R} \\ flo > 1 \quad \mathbf{F}^{\,flo} &\rightarrow \mathbf{W}, \mathbf{F}^{\,flo-1} \\ \mathbf{F}^{\,flo} &\rightarrow \mathbf{W} \end{aligned}$$

A superscript before the symbol denotes a condition that must be fulfilled for the production rule to take effect. A superscript after the symbol denotes an inheritable attribute that is assigned to that particular symbol.

In this case, a Building **B** with the attribute of *flo* number of floors is used as the starting symbol. As per the first production rule, the symbol is rewritten into a Floor **F** with the same value for attribute *flo*, and a terminal roof **R**.

Unlike in Grammar Definition 1, the second production rule now has a condition that determines whether the rule is run; symbol **F**'s attribute *flo* must be higher than one. If the condition is fulfilled, then a set of walls **W** is created, as well as another floor **F** with an attribute *flo* that is equal to the preceding symbol **F**'s *flo*, minus one.

If the second production rule's condition was not fulfilled, then the third production rule is run. Symbol **F** is rewritten into a set of walls **W**, and the grammar comes to an end.

This adapted grammar would allow for a deterministic amount of floors, based entirely on what value is attributed to starting symbol **B**'s *flo* attribute. If, for example, a value of 3 was used for **B**'s *flo*, then the production rules from Grammar Definition 1 would run in the following order:

$$\mathbf{B}^{\,3} \rightarrow \mathbf{F}^{\,3}, \mathbf{R}$$
$$^{3>1}\ \mathbf{F}^{\,3} \rightarrow \mathbf{W}, \mathbf{F}^{\,3-1}$$
$$^{2>1}\ \mathbf{F}^{\,2} \rightarrow \mathbf{W}, \mathbf{F}^{\,2-1}$$
$$\mathbf{F}^{\,1} \rightarrow \mathbf{W}$$

In this instance it can be seen that the second production rule would run twice, with the attributed *flo* value decrementing by one each time. On the third run, the condition for the second production rule would not be fulfilled, and so the program would instead run the third production rule. In this way, if the grammar were run with **B**'s initial *flo* value at 3, a total of three floors would be created.

It ought to be noted that, despite the use of an unorthodox notation, this style of grammar would still be considered context-free, as well as syntaxically and semantically correct. A more standardised form of notation could be created by expanding the condition for every possibility (in this case, an expansion of *flo* if the attribute is equal to one, two, three etc.), but if the boundaries of a condition are not known, then the list of production rules might tend towards infinity. The use of a condition therefore keeps the grammar concise and readable.

This solution effectively addresses the issue of ambiguity within standardised formal grammars. However, another problem is still present: it would be difficult to describe architecture with any degree of accuracy or specificity without further context of what the symbols geometrically represent.

To continue with our example, we could state that wall object **W** represents four adjoining rectangular polygons, and that roof **R** represents a single plane that rests on top. This might be enough information for a human to understand the intended meaning of the grammar, but it is insufficient for an application because key information is missing: the size of each object, and each object's location in space. To fulfil this criterion, additional attributes have to be created.

Grammar Definition 3: Simple building with expanded attributes

---

$$\begin{aligned} \text{Non} - \text{Terminals} &= \{B, F\} \\ \text{Terminals} &= \{W, R\} \\ \text{Start Symbol} &= B \end{aligned}$$

$$\mathbf{B}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{F}\,{}^{w,h,d}_{x,y,z}, \mathbf{R}\,{}^{w,h,d}_{x,y+h,z}$$
$$^{y+1<h}\ \mathbf{F}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{W}\,{}^{w,h,d}_{x,y,z}, \mathbf{F}\,{}^{w,h,d}_{x,y+1,z}$$
$$\mathbf{F}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{W}\,{}^{w,h,d}_{x,y,z}$$

There are now six attributes for every symbol. Attributes w and d represent the relative width and depth of the corresponding object. Attribute h is similar, but for the purpose of concise notation it remains static; the h attribute represents the total number of floors on the building, similar to the no longer used attribute *flo*. Attributes x, y, and z represent the relative position of the corresponding object within three-dimensional Euclidian space. The second production rule has also been altered slightly to better accommodate the new geometric approach; whereas attribute *flo* decremented towards zero, attribute y increments towards the value of h.

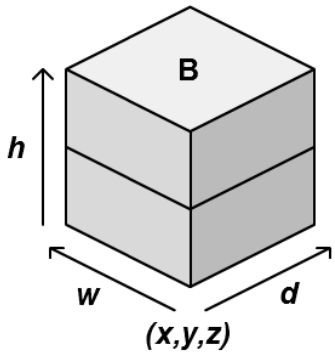

Figure 14: A visual depiction of building B as defined by Grammar Definition 3, with new attributes x, y, z, w, d, and h added.

We can now manually assign the position and the dimensions of the building. In this example, we will specify an x, y, and z value of zero, indicating that the building is being created on the point of origin. We will assign w and d a value of 5, indicating that the building is 5 units wide and long, and we will assign h a value of 3, indicating that there will be 3 floors.

$$\mathbf{B}\,{}^{5,3,5}_{0,0,0} \rightarrow \mathbf{F}\,{}^{5,3,5}_{0,0,0}, \mathbf{R}\,{}^{5,3,5}_{0,0+3,0}$$

$$0+1<3 \quad \mathbf{F}\,{}^{5,3,5}_{0,0,0} \rightarrow \mathbf{W}\,{}^{5,3,5}_{0,0,0}, \mathbf{F}\,{}^{5,3,5}_{0,0+1,0}$$

$$1+1<3 \quad \mathbf{F}\,{}^{5,3,5}_{0,1,0} \rightarrow \mathbf{W}\,{}^{5,3,5}_{0,1,0}, \mathbf{F}\,{}^{5,3,5}_{0,1+1,0}$$

$$\mathbf{F}\,{}^{5,3,5}_{0,2,0} \rightarrow \mathbf{W}\,{}^{5,3,5}_{0,2,0}$$

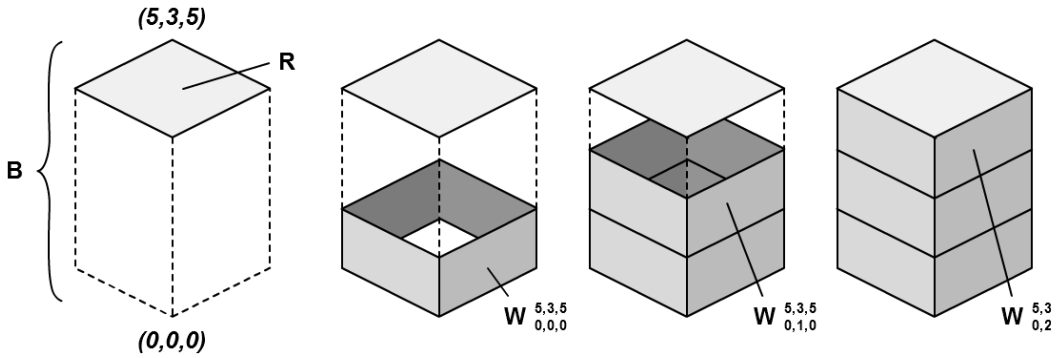

Figure 15: A visual depiction of each stage of the production rules. Roof R is produced first, and then the floors are called recursively, filling in the building B from y=0 through to y=2.

With these attributes assigned, a building is produced on point (0,0,0), and within the boundaries of (5,3,5).

Despite the additional clarity, the new attributes and production rules still do not offer enough information for a programmed implementation. For example, although we know that $\mathbf{W}\,^{5,3,5}_{0,0,0}$ represents a set of four walls with a width and depth of 5 units, we do not know how each wall ought to be represented in terms of triangular or quadrilateral polygons. Therefore, for a truly specific grammar, another layer of production rules could be implemented.

Grammar Definition 4: Simple building with polygonal symbol

$$
\begin{aligned}
\text{Non} - \text{Terminals} &= \{B, F, W, R\} \\
\text{Terminals} &= \{P\} \\
\text{Start Symbol} &= B
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{B}\,^{w,h,d}_{x,y,z} &\rightarrow \mathbf{F}\,^{w,h,d}_{x,y,z}, \mathbf{R}\,^{w,h,d}_{x,y+h,z} \\
y{+}1 < h \quad \mathbf{F}\,^{w,h,d}_{x,y,z} &\rightarrow \mathbf{W}\,^{w,h,d}_{x,y,z}, \mathbf{F}\,^{w,h,d}_{x,y+1,z} \\
\mathbf{F}\,^{w,h,d}_{x,y,z} &\rightarrow \mathbf{W}\,^{w,h,d}_{x,y,z} \\
\mathbf{W}\,^{w,h,d}_{x,y,z} &\rightarrow \mathbf{P}\,^{x,y,z,x+w,y,z}_{x+w,y+1,z,x,y+1,z} \\
&\rightarrow \mathbf{P}\,^{x,y,z+d,x+w,y,z+d}_{x+w,y+1,z+d,x,y+1,z+d} \\
&\rightarrow \mathbf{P}\,^{x,y,z,x,y,z+d}_{x,y+1,z+d,x,y+1,z} \\
&\rightarrow \mathbf{P}\,^{x+w,y,z,x+w,y,z+d}_{x+w,y+1,z+d,x+w,y+1,z} \\
\mathbf{R}\,^{w,h,d}_{x,y,z} &\rightarrow \mathbf{P}\,^{x,y,z,x+w,y,z}_{x+w,y,z+d,x,y,z+d}
\end{aligned}
$$

In this definition, a new terminal symbol **P** has been added, representing a quadrilateral polygon within Euclidian space. It has 12 attributes, representing the x, y, and z coordinates for each of the polygon's four vertices (i.e. $\mathbf{P}\ {}^{x1,y1,z1,x2,y2,z2}_{x3,y3,z3,x4,y4,z4}$).

To accommodate for this change, terminal symbols **W** and **R** have been changed into non-terminals. The two symbols have received their own production rules that create sets of terminal **P** symbols, generating polygonal representations of the building elements.

Despite the additional clarity, it was decided upon to not use the polygonal level of detail for the remainder of the grammar definitions within this thesis. This decision was made for the purpose of legibility. As the complexity of modelled buildings increases, the complexity and length of the notation must also increase. An especially detailed structure may require several pages of rules, which would make the grammar fairly incomprehensible. We must aim to strike a balance between technical detail and user legibility.

Therefore, our grammar definitions will take the form laid out by Grammar Definition 3. Conditional statements for a production rule are notated as a superscript on the left-hand side. Attributes of a symbol are notated as subscripts and superscripts to the right of the defined symbol. The attributes of a non-terminal symbol are inherited by any called production symbols, which allows for a hierarchy of geometric shapes to be constructed.

With these rules laid out, we can proceed to start defining Vitruvian architecture.

3.2.5.2 Temple

The first building to be defined using the novel shape grammar syntax, and the building to be described in the greatest depth, was the Roman temple. This structure was chosen first due to its immediate recognisability, its prevalence across a large number of Roman settlements, and its integral significance in everyday Roman life. Additionally, the structure features certain elements, such as columns and a typical slanted roof, which can be reused across multiple other types of buildings.

Vitruvius covered the architectural design of Roman temples in extensive detail. Two books are devoted to the construction process (De Architectura III and IV), with the majority of the text focussing on the different orders of columns, the proportions of the various architectural elements, and the considerations that must be made for the different styles of temple building.

Several features are prevalent across the majority of temples described by Vitruvius. The structures frequently feature a central building with an encompassing wall, called a cella. A colonnade would usually be found at the front portico of the building, four or six columns across, and one to four columns deep. The temple would often be oriented such that the main door of the cella faces eastwards or towards the forum, or if that is not feasible, facing the passing road.

However, a large number of variations are possible in temple construction. Vitruvius acknowledges that the dimensions of the architectural elements can differ significantly, depending on the local style, the deity the temple is dedicated to, and the personal preferences of the architect. For example, temples can be built with or without a podium; they can feature steps on all four sides or just the front; and they can feature prominently-displayed altars or no altars at all. Vitruvius even describes peculiarities like circular temples, which often contained no internal cella (De Architectura IV, 8).

When we came across cases of possible variation, we attempted to conform to Vitruvius' recommended standard. If no such recommendation existed or if key information was missing, then we attempted to make an educated guess based upon the context of the surrounding information.

Ironically, many of the temples that Vitruvius cites for the purpose of illustrative example, such as his comparisons to the three temples of Fortuna (De Architectura III, 2, 3), no longer exist. When a key measurement is not provided in De Architectura, we therefore had to draw estimates based on archaeological record, or on existing temples that appear to conform to the Vitruvian ideal.

Podium

The raised supporting foundation of the temple, known as the base or podium, consists of a simple platform that acts as a floor for the cella and colonnades, and two additional platform structures that function as bannisters to the stairs. Temple podiums frequently feature top surfaces that extrude outward from the base, but Vitruvius make no mention of the ideal level of protrusion for these details. Nonetheless, we can estimate the dimensions of the protruding sections from examples in the archaeological record.

Geometrically, we can consider the podium to consist of three cuboids, with three flatter and wider cuboids resting on top. Vitruvius explicitly stated that "the length of a temple must be twice its width" (De Architectura IV, 4), and since the temple's podium runs across the entirety of the temple's dimensions, this rule also applies

here (Figure 13). We are not told of the ratio between the length of the bannisters and the rest of the podium, but this can be estimated from the given dimensions of the cella and stairway.

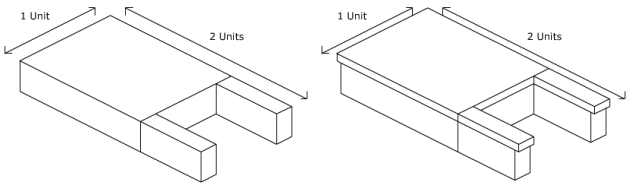

Figure 16: The relative proportions of a temple cella podium, without and with the top layer.

Stairway

The stairway that lies at the front of the temple simply consists of a series of steps. Vitruvius noted that the step depth ought to be between 1.5 and 2 feet, that the step height ought to be between 0.75 and 0.83 feet, and that the steps ought to be consistent in size (De Architectura III, 4). From these measurements, we can infer a step depth to height ratio of approximately 2 to 1, and so this was the ratio adopted for use in the application. By placing two rectangular polygons together in this ratio, a single step can be created that, when replicated and translated, create the entire stairway (Figure 14).

Interestingly, Vitruvius noted that the number of steps in front of a temple ought to be odd, since that would ensure that "the right foot, which begins the ascent, will be that which first alights on the landing of the temple" (De Architectura III, 4). This is simple to account for in the application by adjusting the height of every step until the requirement has been reached.

One additional note is that some temples featured altars at the base of the stairway. Although the size, shape, and detail of the altar varied between temples, the overall structure could be geometrically represented as one smaller cuboid resting upon a larger one. Due to the level of variety among altars and the lack of information on their measurements in Vitruvius' writings, the stairway altar was not adapted into the application.

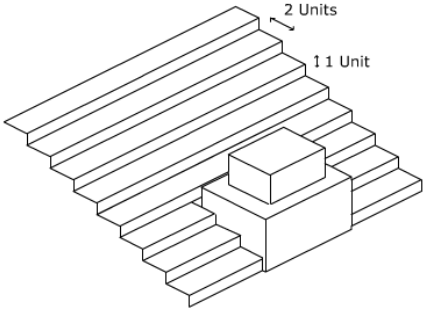

Figure 17: The temple stairway and altar.

Cella

The cella is the inner room of a temple, often housing a statue of a deity or a plinth for the purpose of votive offerings. They varied in size and shape – Vitruvius acknowledged and described variations where there may be multiple cella rooms or no surrounding walls at all (De Architectura IV, 8) – but they typically consisted of a single rectangular room with a single entranceway.

Vitruvius described the outer cella wall as being "in length, one fourth more than the breadth" (De Architectura IV, 4). He did not specify the ideal thickness of the cella wall, but acknowledged that the thickness "must depend on the magnitude of the work". To adapt this description, we created a "wall width" variable that adjusted in accordance to the temple's length. As such, the wall thickness, and the perceived structural integrity, increases relative to the temple size.

Vitruvius specified the dimensions of the front doorway through relative comparisons. He wrote that, if we assume the height of the cella to be 3.5 units, then the height of the doorway ought to be 2.5 units. Additionally, if we assume the doorway height to be 12 units, we can define the doorway width as being 5.5 units (De Architectura IV, 6). Put together, we can therefore state that the ratio of the door width to door height to cella height is approximately 2.29:5:7 (Figure 15). This ratio can then be used to calculate the in-application dimensions of the doorway, as long one dimension is already known.

Breaking this information down into geometric primitives, we can consider the cella to consist of three cuboids that encompass the rear and sides of the structure, and an additional three cuboids that fit the aforementioned dimensions to form the front. An additional specially-textured cuboid can be used to represent the door itself, if desired.

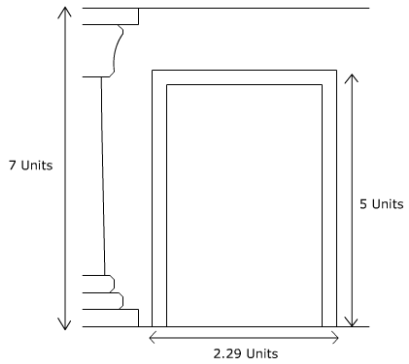

Figure 18: The relative proportions of a temple cella doorway.

Columns

Although columns were a notable feature of all Roman temples, the layout and form of the structures varied between buildings. Factors like the size and dimensions of the temple, the purpose of the temple, the time period, and the local style, all dictated how the columns would be laid out and shaped.

Vitruvius did not voice a preference on the recommended number of columns upon a temple's floor plan, but instead defined a set of possibilities for how columns could be arranged relative to the cella, and listed a set of examples (De Architectura III, 2). He made note of how the columns upon the portico could be tetrastyle, hexastyle, or octastyle, depending on whether there were four, six, or eight columns across the temple's front. He also noted that the columns that flanked the cella could be single or double placed, and that the columns at the rear could either mirror the front of the temple or be a simple single-line colonnade.

Vitruvius also outlined a set of classifications for defining the distance between columns, relative to the column width. He specified that intercolumniation could be pycnostylos (that is, the columns are placed one-and-a-half base diameters apart), systylos (two diameters apart), eustylos (two-and-a-quarter diameters apart), diastylos (three diameters apart), or araeostylos (more than three diameters apart). Vitruvius lamented that the tighter styles of intercolumniation, despite being common at the time, were flawed as they did not allow matrons to "pass the intercolumniations arm in arm". He proposed that the eustylos standard was preferable, due to its "respect of convenience … beauty, and strength" (De Architectura III, 3). Consequently, this was the style chosen for the intercolumniation

of the generated temple colonnades; the columns in the application are positioned two-and-a-quarter diameters apart.

With these considerations in mind, and using the measurements imposed by the boundaries of the insula and the other temple elements, we decided that a double-ranked tetrastyle colonnade would be most appropriate for the front portico of the temple.

With the column positions sited, the column geometry is ready to be defined. Vitruvius wrote at length of the proportions of Roman columns, and made particularly explicit note of the relative dimensions of the Ionic order. He stated that the diameter at the base of an Ionic column is equal to one seventh of the column height, and the diameter at the top of the column is equal to three quarters of the diameter at the bottom. Additionally, the column base height is equal to half the diameter at the bottom of the column, and the whole column height should equal one third of the temple width (De Architectura III, 5). The specificity of these rules makes them convenient for our implementation; as long as one element is known – in this case, the temple width – the other measurements can be calculated.

An important consideration at this stage was the amount of detail to contain in the procedurally generated column geometry. If a particularly accurate representation of Roman column geometry were desired, then a large number of polygons would be needed to achieve the necessary level of detail. This would result in a complex and potentially more lifelike model, but it could reduce the functionality of the application if a large number of columns are to be displayed on-screen simultaneously. Conversely, a column object with a low number of polygons would help to ensure that the application can run in real-time, but it may not be complex enough to reflect the defined Vitruvian measurements.

A compromise must therefore be reached. After experimenting with various setups, we settled on a geometrically simple structure, featuring a shaft of 8 rectangular polygons (or 16 triangular polygons), and a total polygon count of 76. The grooves in the shaft can simply be represented through the use of a texture.

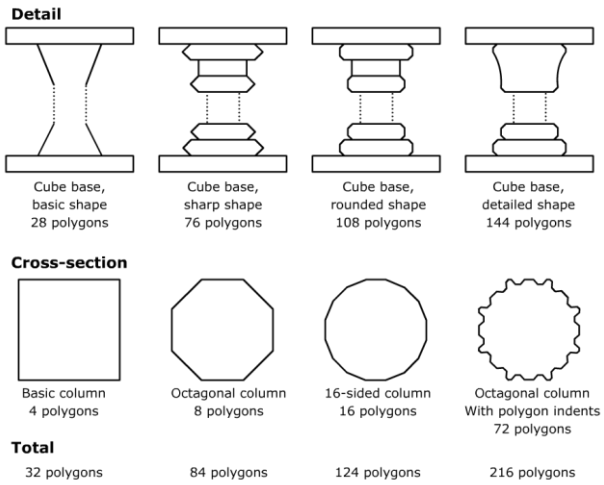

Figure 19: Four potential levels of detail for a column of the Doric or Corinthian order. The second column image was the one implemented into our application.

Grammar Definition 5: Roman Temple

$$\text{Non} - \text{Terminals} = \begin{Bmatrix} S - Side, St - Stairway, P - Podium, \\ A - Altar, Ce - Cella, Cn - Colonnade, T - Temple \end{Bmatrix}$$

$$\text{Terminals} = \begin{Bmatrix} G - Ground, Co - Column, R - Roof, \\ S - Step, B - Block, D - Door \end{Bmatrix}$$

$$\text{Start Symbol} = T - Temple$$

$$\mathbf{T}_{x,y,z}^{w,h,d} \rightarrow \mathbf{P}_{x,y-2,z}^{w,2,d}, \mathbf{R}_{x+1,y,z+2}^{w-2,h,d}, \mathbf{St}_{x,y,z}^{w,h,d}, \mathbf{Ce}_{x,y,z}^{w,h,d}, \mathbf{G}_{x,y,z}^{w,h,d}$$

$$\mathbf{St}_{x,y,z}^{w,h,d} \rightarrow \mathbf{S}_{x,y,z+d}^{w,0.2,0.4} \dots \mathbf{S}_{x,y+h,z}^{w,0.2,0.4}, \left(\mathbf{A}_{x+4,y,z+0.5}^{w,h,d}\right)$$

$$\mathbf{P}_{x,y,z}^{w,h,d} \rightarrow \mathbf{B}_{x,y,z}^{1,2,3}, \mathbf{B}_{w-1,y,z}^{1,2,3}, \mathbf{B}_{x,y,3}^{w,2,d-3}$$

$$\mathbf{A}_{x,y,z}^{w,h,d} \rightarrow \mathbf{B}_{x,y,z}^{2,1,1}, \mathbf{B}_{x+0.25,y+1,z+0.25}^{1.5,1,0.5}$$

$$\mathbf{Ce}_{x,y,z}^{w,h,d} \rightarrow \mathbf{B}_{x,y,z}^{1,h,d}, \mathbf{B}_{w-1,y,z}^{1,h,d}, \mathbf{B}_{x,y,d-1}^{w,h,1}, \mathbf{B}_{x,y,z}^{4,h,1}, \mathbf{B}_{x+6,y,z}^{4,h,1},$$

$$\rightarrow \left(\mathbf{D}_{x+4,y,z+0.5}^{w,h,d}\right), \mathbf{Cn}_{x,y,z}^{w,h,d}, \mathbf{Cn}_{x+w,y,z}^{w,h,d}, \mathbf{Cn}_{x,y,z+d}^{w,h,d}$$

$$\mathbf{Cn}_{x,y,z}^{w,h,d} \rightarrow \mathbf{Co}_{x,y,z}^{w,h,d} \dots \mathbf{Co}_{x+w,y,z}^{w,h,d}$$

Having completed the architecture of the temple, we can now implement grammars for the remaining structures. By making use of a hierarchy of elements for one structure, we are now able to reuse desired elements for the remaining buildings. For example, the columns of the temple require little adjustment to be applied to a forum; only the grammar implementation needs to be altered.

3.2.5.3 Forum

Roman forums, the central squares that functioned as open-air gathering places, were an integral feature of Roman settlements. Although they were not buildings in the traditional sense, forums were still composed of a set of typical building elements, and so they could be defined through the same set of grammatical rules as the other architectural structures in this project.

Vitruvius noted that Greek forums were typically close-set, double-width colonnades encompassing square platforms, but he asserted that this was an inappropriate setting for Roman cities where the forum was occasionally used for public displays like gladiator shows (De Architectura V, 1). He instead proposed that Roman forums ought to have a width that is two thirds of its length, and that the surrounding colonnade ought to be widely spaced and two floors tall.

The geometry involved in the procedural rendering of a forum is simple, as the most important feature tends to be the communal space itself. Nonetheless, a problem became apparent when attempting to define the forum's dimensions within the allotted insula: the entrance of the neighbouring temple encroached on the forum's space. Vitruvius made no mention of the influence that a neighbouring temple can have on the dimensions of a forum, but did specify that a neighbouring basilica or market hall ought to take up one third of a forum's floor space; the colonnade can occupy the remaining oblong area (De Architectura V, 1). By treating the temple as a basilica, the dimensions could be adjusted to ensure that the forum colonnade and the temple entranceway could cohabit without causing overlapped geometry.

Grammar Definition 6: Roman Forum

$$\begin{aligned} \text{Non} - \text{Terminals} &= \{Cd - Colonnade\} \\ \text{Terminals} &= \{G - Ground, C - Column\} \\ \text{Start Symbol} &= F - Forum \end{aligned}$$

$$\mathbf{F}\,{}^{w,2,d}_{x,y,z} \rightarrow \mathbf{Cd}\,{}^{w,h,d}_{x,y,z}, \mathbf{Cd}\,{}^{w,h,d}_{x+w,y,z}, \mathbf{Cd}\,{}^{w,h,d}_{x,y,z+d}, \mathbf{Cd}\,{}^{w,h,d}_{x+w,y,z+d}, \mathbf{G}\,{}^{w,h,d}_{x,y,z}$$

$$y{+}1 < h \quad \mathbf{Cd}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{C}\,{}^{w,h,d}_{x,y,z} \ \ldots\ \mathbf{C}\,{}^{w,h,d}_{x+w,y,z}, \mathbf{Cd}\,{}^{w,h,d}_{x,y+1,z}$$

3.2.5.4 Theatre

Early Roman theatres were timber structures; it was only during the time of Vitruvius that permanent structures became increasingly popular, with architecture based upon earlier Greek examples. The traditional Roman theatre consisted of a stage area, which was often a covered building, and an inner seating area that formed a semi-circle. The outside would often consist of tall, rounded archways, with columns adjacent to or embedded in the outer wall.

Vitruvius defined the angle formed by the semi-circular section of the theatre by noting that four equilateral triangles, each with points on the centre of the *orchestra* and perimeter of the theatre, could fit within the theatre's arc (*De Architectura* V, 6). Taken literally, this would imply that the semi-circular section of Roman theatres ought to form a 240 degree arc. Vitruvius contrasted this with Greek theatres, where he compared the angle to that of three squares, or 270 degrees. This appears consistent with the archaeological record. Although the specific angle varies, many

Roman theatres, such as the theatre at Bosra, Syria, appear to have an orchestra angle of close to 240 degrees, whereas Greek theatres such as the one in Epidaurus, Greece, are notably more arced (De Malsche, et al., 1983).

Vitruvius also offered measurements for the staging area. For example, he specified that, "the length of the scene must be double the diameter of the orchestra," and that, "the height of the podium, or pedestal … is a twelfth part of the diameter of the orchestra" (*De Architectura* V, 6). Vitruvius acknowledged that it would be inevitable that each theatre would be built with slightly different dimensions due to restrictions of circumstance, but he repeatedly stressed the importance of symmetry for the purposes of aesthetic beauty and good acoustics.

Many of the elements for the theatre, such as the columns and seating area, can be borrowed from the architectural elements used in the construction of the temple. One element that is new to our shape grammars is the archway, which must be repeatedly called at various angles and at two to three heights in order to form the outer wall of the theatre structure.

Grammar Definition 7: Theatre

$$\begin{aligned} \text{Non}-\text{Terminals} &= \{St-Stairway, S-Side, T-Theatre\} \\ \text{Terminals} &= \{A-Archway, C-Column, Se-Step, Sa-Stage\} \\ \text{Start Symbol} &= T-Theatre \end{aligned}$$

$$\begin{aligned} \mathbf{T}_{x,y,z}^{w,h,d} &\rightarrow \mathbf{S}_{x,y,z}^{w,h,angle}, \mathbf{Sa}_{x,y,z}^{w,h,d} \\ angle < 240 \quad \mathbf{S}_{x,y,z}^{w,h,angle} &\rightarrow \mathbf{S}_{x,y,z}^{w,h,angle+30}, \mathbf{Ar}_{x,y,z}^{w,h,d}, \mathbf{C}_{x,y,z}^{w,h,d}, \mathbf{St}_{x,y,z}^{w,h,d} \\ &\rightarrow \mathbf{Ar}_{x,y+h,z}^{w,h,d}, \mathbf{C}_{x,y+h,z}^{w,h,d} \\ \mathbf{St}_{x,y,z}^{w,h,d} &\rightarrow \mathbf{Se}_{x,y,z+d}^{w,0.2,0.4} \ \ldots \ \mathbf{Se}_{x,y+h,z}^{w,0.2,0.4} \end{aligned}$$

3.2.5.5 Amphitheatre

At the time of Vitruvius' writing, amphitheatres were considered to be something of a novelty, not a prevalent piece or architecture (*De Architectura* I, 7). However, enough details have been left through public record and archaeological remains for us to gather accurate proportions of a typical structure. Amphitheatres vary in shape and size, but always took the form of a fully-enclosed seating and stairway area. Some were designed to rest upon hillsides, where others stood as monumental, vaulted structures. For the purposes of this project, only the latter type was chosen to

be implemented. Vitruvius states that the outer length to width ratio of the amphitheatre ought to be 1.2:1, but that the inner ratio ought to be 1.5:1, creating a notable oval shape. Aside from the difference in structure shape and the lack of a separate staging area, the physical differences between theatres and amphitheatres are minor. This allows for many of the same production rules to be borrowed from the theatre grammar.

Grammar Definition 8: Amphitheatre

$$\begin{aligned} \text{Non} - \text{Terminals} &= \begin{Bmatrix} St - Stairway, S - Side, \\ A - Amphitheatre \end{Bmatrix} \\ \text{Terminals} &= \{C - Column, Se - Step, Ar - Archway\} \\ \text{Start Symbol} &= A - Amphitheatre \end{aligned}$$

$$\begin{aligned} \mathbf{A}\,{}^{w,h,d}_{x,y,z} &\rightarrow \mathbf{S}\,{}^{w,h,angle}_{x,y,z} \\ angle < 360 \quad \mathbf{S}\,{}^{w,h,angle}_{x,y,z} &\rightarrow \mathbf{S}\,{}^{w,h,angle+30}_{x,y,z}, \mathbf{Ar}\,{}^{w,h,d}_{x,y,z}, \mathbf{C}\,{}^{w,h,d}_{x,y,z}, \mathbf{St}\,{}^{w,h,d}_{x,y,z} \\ &\rightarrow \mathbf{Ar}\,{}^{w,h,d}_{x,y+h,z}, \mathbf{C}\,{}^{w,h,d}_{x,y+h,z} \\ \mathbf{St}\,{}^{w,h,d}_{x,y,z} &\rightarrow \mathbf{Se}\,{}^{w,0.2,0.4}_{x,y,z+d} \ \ldots\ \mathbf{Se}\,{}^{w,0.2,0.4}_{x,y+h,z} \end{aligned}$$

3.2.5.6 Governmental Building

A governmental building grammar was designed, both to fulfil the requirements for a basilica, the extravagant Roman public court building, and to serve the generation of the numerous "generic" governmental buildings that a town is likely to contain.

Vitruvius explicitly described the measurements of a basilica. Most notably, he commented that, "The columns of basilica are to be of a height equal to the breadth of the portico, and the width of the portico one-third of the space in the middle", and that "The portico … is twenty feet wide … The height of the columns, including the capitals, is fifty feet" (*De Architectura* V, 1). Put together, this provides enough information to form our shape grammar, borrowing elements from the previously defined temple where necessary.

To create the simpler, generic governmental building structure from the basilica, the only changes to be made were to reduce the structure's dimensions, and to remove the more superfluous elements of the basilica, such as the double-height columns

and the portico. This can be performed semi-randomly, producing a range of structures that still resemble a public building.

Grammar Definition 9: Governmental Building

$$\text{Non} - \text{Terminals} = \{W - Walls, S - Side, F - Floor, G - Governmental\}$$
$$\text{Terminals} = \{D - Doorway, Wi - Window, R - Roof, C - Columns\}$$
$$\text{Start Symbol} = G - Governmental$$

$$\mathbf{G}\,{}^{w+rand(0,3),rand(1,3),d+rand(0,3)}_{x+rand(0,3),y,z+rand(0,3)} \rightarrow \mathbf{F}\,{}^{w,h,d}_{x,y,z}, \mathbf{R}\,{}^{w,h,d}_{x,y+h,z}$$

$$y+1 < h \quad \mathbf{F}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{W}\,{}^{w,h,d}_{x,y,z}, \mathbf{F}\,{}^{w,h,d}_{x,y+1,z}$$

$$\mathbf{F}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{W}\,{}^{w,h,d}_{x,y,z}$$

$$\mathbf{W}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{S}\,{}^{w,h,d}_{x,y,z}, \mathbf{S}\,{}^{w,h,d}_{x+w,y,z}, \mathbf{S}\,{}^{w,h,d}_{x,y,z+d}, \mathbf{S}\,{}^{w,h,d}_{x+w,y,z+d}$$

$$rand(0,10)==1 \quad \mathbf{S}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{D}\,{}^{w,h,d}_{x,y,z}$$

$$\mathbf{S}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{Wi}\,{}^{w,h,d}_{x,y,z}, (\mathbf{C}\,{}^{w,h,d}_{x,y,z})$$

3.2.5.7 Villa

Both the Roman villas owned by nobles and the town-houses of the lower classes are formed of previously-defined architectural elements; we only need to know the dimensions and the layout of the needed elements.

Vitruvius documented the size and style considerations for the interior of villas extensively, and made note of the differences between villa rustica, country houses, and villa urbana, suburban residences (*De Architectura* VI, 1-8). He noted that the atrium, the villa's internal courtyard, can lie between thirty and eighty feet in length, and that the atrium's width and the size of the neighbouring rooms must be adjusted to accommodate this large potential length deviation. He also offered measurements for various rooms, such as, "the cloister is transversely one third part longer than across", and "the length of a triclinium is to be double its breadth". Vitruvius stops short of explicitly defining the size of the villa's perimeter, but we can draw estimates based on the given information.

In the same way that generic governmental buildings can be formed by resizing and simplifying the architecture of the basilica, we are able to create humbler residential structures by removing the villa's atrium and adjusting the building size with some degree of randomness.

Grammar Definition 10: Villa

$$\text{Non} - \text{Terminals} = \{W - Walls, S - Side, F - Floor, V - Villa\}$$
$$\text{Terminals} = \{D - Doorway, Wi - Window, R - Roof, A - Atrium\}$$
$$\text{Start Symbol} = V - Villa$$

$$\mathbf{V}\,{}^{w+rand(0,3),rand(1,2),d+rand(0,3)}_{x+rand(0,3),y,z+rand(0,3)} \rightarrow \mathbf{F}\,{}^{w,h,d}_{x,y,z}, \mathbf{R}\,{}^{w,h,d}_{x,y+h,z}, \left(\mathbf{A}\,{}^{w,h,d}_{x+w,y,z}\right)$$

$$y+1 < h \quad \mathbf{F}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{W}\,{}^{w,h,d}_{x,y,z}, \mathbf{F}\,{}^{w,h,d}_{x,y+1,z}$$

$$\mathbf{F}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{W}\,{}^{w,h,d}_{x,y,z}$$

$$\mathbf{W}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{S}\,{}^{w,h,d}_{x,y,z}, \mathbf{S}\,{}^{w,h,d}_{x+w,y,z}, \mathbf{S}\,{}^{w,h,d}_{x,y,z+d}, \mathbf{S}\,{}^{w,h,d}_{x+w,y,z+d}$$

$$rand(0,10)==1 \quad \mathbf{S}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{D}\,{}^{w,h,d}_{x,y,z}$$

$$\mathbf{S}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{Wi}\,{}^{w,h,d}_{x,y,z}$$

$$\mathbf{A}\,{}^{w,h,d}_{x,y,z} \rightarrow \mathbf{W}\,{}^{1,h,d}_{x,y,z}, \mathbf{W}\,{}^{w,h,1}_{x,y,z}, \mathbf{W}\,{}^{1,h,d}_{x,y,z+d}, \mathbf{W}\,{}^{w,h,1}_{x+w,y,z}$$

3.3 RENDERING ENGINE

3.3.1 CREATING THE DIRECTX CITY FILE

Converting the formal shape grammars defined in section 3.2.5 into C++ code requires a simple, but methodical process. Each unique symbol in each of the formal grammars is assigned a function. Accompanying symbol attributes – the superscripts and subscripts following each symbol – are latched to the corresponding function through appropriately typed function parameters. Conditions – the superscripts preceding a symbol – are implemented through conditional "if" statements contained within the function bodies, or an appropriate loop or switch-case statement where necessary. As such, just as the non-terminal symbols called other symbols in the defined grammars, the architectural functions call other functions in the programming code. A hierarchy has been created.

However, the functions representing terminal symbols must make note of the geometry that they represent. This involves taking the function parameters, and parsing the values to a set of arrays that represent vertex points, polygon points, and texture references.

It would be entirely possible to use the data created in the generation process to render polygons directly to the screen, as the data is essentially stored in a set of vertex, polygon, and texture arrays. However, the decision was made to write the data to a DirectX (.x) file, which could then be read and output to the screen. The reason for this is twofold.

First, the implemented method of storing and retrieving data from a custom vertex array is likely to be less efficient than Microsoft's implemented method. Rather than spending time experimenting to find an optimal way of reading a vertex array, it would be more prudent to rely on a tried and tested method.

Second, storing the data in a widely-accepted model file format allows for the possibility of exporting or sharing the model. If the user desires the model file for a game, animation, or their own simulation, then passing on the .x file along with the necessary textures is a simple process.

DirectX files were chosen over other model file types (such as .3DS, .MAX, or .C4D), primarily due to the unencrypted, text encoded nature of the file format. The Object file type (.obj) is another commonly-used file type that uses unencrypted, text encoded data, and so it was considered for this application. The difference between the two file types is marginal, in terms of both implementation and efficiency. The

decision to use the DirectX file type over the Object file type was therefore made out of convenience, as the DirectX 9 framework more readily supports the implementation of .x meshes over .obj meshes.

An important choice at this stage of the application was whether it would be more beneficial to write to a single mesh containing the entire city, or write to multiple meshes that contain each building. Multiple meshes can either be written to within a single DirectX file (in the form of multiple frames), or spread across multiple files. Table 2 demonstrates the comparative advantages of the use of single or multiple meshes in this scenario, with respect to application efficiency, the generated model file size, and the potential for culled geometry.

	Single Mesh	Multiple Meshes
Efficiency	Even with a large number of textures, reading a single mesh file is an efficient process. The only issue is the possibility of a stack overflow when writing large quantities of data, but taking care when allocating and deallocating memory would prevent this.	Generally, multiple meshes require multiple parses of the same texture, and may use multiple swaps of the z-buffer. At its extreme (such as in a city with over one thousand buildings), this could cause efficiency issues.
File Size	A single file or mesh is more portable than multiple meshes. However, the model file is likely to be larger in size.	If similar buildings are repeated, it may be possible to reuse the same model mesh or model frame for multiple buildings. This may or may not decrease the RAM usage, but it would significantly decrease the size of the model files used.
Culling	Since every building is a part of a single object, cities must be culled on a per-polygon basis. This is not necessarily less efficient, but it does require more forethought to implement effectively.	Since each building has its own frame or object, buildings can be culled on a per-building basis. Further culling parameters may be necessary, but this would be a simple process.

Table 2: A comparison of the relative advantages and disadvantages of using a single mesh or multiple meshes within a DirectX application.

At a glance, it appears that writing to multiple meshes is the logical choice as this would be an easier to implement. Additionally, the ability to transform each building on its own matrix at any point in the rendering stage, not just the city generation stage, could allow for a lot more possibilities. However, efficiency is an important factor, especially when we consider that this application could potentially reach 100,000+ polygons in size, and must be able to run in real-time. If multiple meshes

would require multiple checks or multiple swaps on the Z-buffer, where a single mesh would only require one, then this could become an important, difficult-to-fix issue in the later stages of programming.

Having established that we would be writing to a single mesh contained in a single DirectX file, a method had to be established of exporting the virtual city data. For the sake of speed, ease of use, and readability, two text files were created that contained sets of information that would be persistent across multiple runs of the application.

The first (named "XFilePart1.txt" in this application) contains a list of headers, structures, and templates, and a standard frame transformation matrix. The purpose of this file is to reduce the amount of redundant code; these elements are likely to be identical between the different .x files created for different cities, and hard-coding the lines into the application itself would

The second file (named "XFilePart2.txt") contains a list of the textures used in the application, along with a set of normal that are frequently referenced by the mesh geometry. The purpose of this file is to allow for quick alterations of the currently used texture, without having to recompile the code. For example, changing the name of “brick.jpg” to “redbrick.png” would successfully alter the texture used for the walls, assuming that the relevant file is present in the correct directory.

A chronological list of the model-writing process follows.

1. Create and name a new .x file - in this case, we named it "CityModel.x".
2. Parse the contents of XFilePart1.txt, line by line, to CityModel.x via a simple loop.
3. Parse every vertex point that was generated to CityModel.x, again via a simple loop.
4. Parse every polygon that was generated to CityModel.x. A loop very similar to the loop in step 3 is used.
5. Parse the contents of XFilePart2.txt to CityModel.x.
6. Parse the normal direction of every vertex to CityModel.x.
7. Parse the mesh coordinates of every polygon to CityModel.x (in this case, (0,0) to (1,1) was parsed, unless specifically stated otherwise).
8. Save and close the file.

The result of this is a fairly large unencoded file (20,614 KB for a 200 building city) containing a single model file of all the entire generated architecture, including the necessary texture and normal map data. The vertex, polygon, and texture arrays that we used for the city generation can then be dropped; their purpose has been fulfilled, and now the arrays only take up memory space with no reason. The newly-created DirectX model file can then be loaded into the program, and rendered through drawing calls native to the DirectX graphics library.

3.3.2 *LIGHTS, CAMERA, SKYBOX*

A simple two light set-up was created, consisting of a neutral-toned ambient light, and a brighter, diffuse and specular-enabled directional light. A conventional skybox was also created for the purpose of encouraging an immersive experience.

In order to allow the user to navigate the city, a moveable camera was set up. The mouse was linked to the camera's rotation, and the A, S, D, and W keys were linked to the camera's relative position. Together, this provides a first-person navigation experience that many people who play videogames would find familiar.

In addition, two navigational modes were set up to allow for the viewing of the city from two perspectives: an eye-level mode, and a free-roaming mode. When in eye-level mode, the camera's Y-axis is linked to the terrain's Y-axis at the camera's current position. In this way, the camera can navigate up and down hills, and is restricted from floating away from the floor; the user is given the experience of a civilian. When in free-roaming mode, the camera is not bound to the floor, and the user can fly into the sky by using the A, S, D, and W keys. This allows the user to see the city from a bird's eye view, if desired.

The M and the N keys were dedicated to the purpose of switching between the two modes.

4 RESULTS

4.1 CITY POSITION

To test the effectiveness of the implemented city positioning algorithm, the three discussed positioning methods of section 3.3.1 were applied to a selection of heightmaps. The heightmaps were created by generating Perlin noise in image editing software GIMP, and then making manual adjustments to achieve the desired output.

The three heightmaps were designed for the explicit purpose of demonstrating each city positioning method's advantages and disadvantages, as well as showing situations where the methods produce near-identical results.

The first heightmap is a relatively flat surface. There is no land below the water line, and therefore no out of bounds areas. There are hills – the highest point lies towards the top-right of the map – but the gradients across the map can be considered to be reasonably shallow.

The second heightmap tested features a lake at the centre, caused by a set of vertices that dip below the water line. Consequently, the centremost points of the map are considered to be out of bounds. The rest of the map is fairly flat.

The third heightmap features two corners that dip below the waterline, and two corners that are quite raised, creating a slope. Consequently, the majority of the map is on a steep gradient, and large parts might be considered unusable.

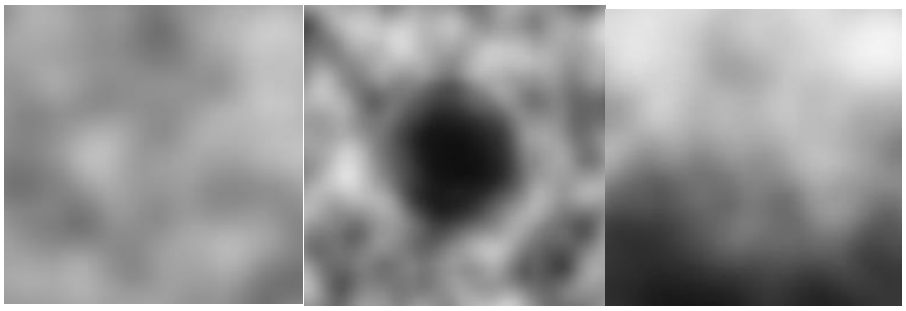

Figure 20: The three heightmaps that were used in the testing of the city position algorithm. The darkest areas on the second and third image lie underwater.

In order to demonstrate the various results that could be produced by the weighed factor formula (section 3.2.1.3), two variations of the method were used: one where there is an emphasis on the centremost point of the map, and one where there is an

emphasis on the highest point of the map. The weighted factor formula was also used for the purpose of notating the four tested city placement algorithms, demonstrated as follows.

Method 1: Centremost point selection

```
PV = (Cd*1) + (Wd*0) + (Rd*0) + {[0*G]*[0*(MH-H)]}
PV = Cd
```

Method 2: Factor formula (unweighted)

```
PV = (Cd*1) + (Wd*1) + (Rd*1) + {[1*G]*[1*(MH-H)]}
PV = Cd + Wd + Rd + [G * (MH - H)]
```

Method 3: Weighted factor formula (center)

```
PV = (Cd*2) + (Wd*1) + (Rd*1) + {[1*G]*[1*(MH-H)]}
PV = Cd*2 + Wd + Rd + [G * (MH - H)]
```

Method 4: Weighted factor formula (height)

```
PV = (Cd*1) + (Wd*1) + (Rd*1) + {[1*G]*[2*(MH-H)]}
PV = Cd + Wd + Rd + {G * [2*(MH-H)]}
```

Where PV represents the Point Value that is measured on every integer coordinate upon a heightmap, Cd represents the distance to the centre of the map, Wd represents the distance to the nearest body of water, Rd represents the distance to the nearest road, G represents the gradient of the land at the current point, MH represents the maximum height upon the heightmap, and H represents the height at the current coordinate.

When implemented into the heightmaps shown in Figure 17, the formulae produced cities upon a range of locations. The sited locations can be seen visually upon the relevant heightmaps in Figure 18, or in tabulated coordinate form in Table 3.

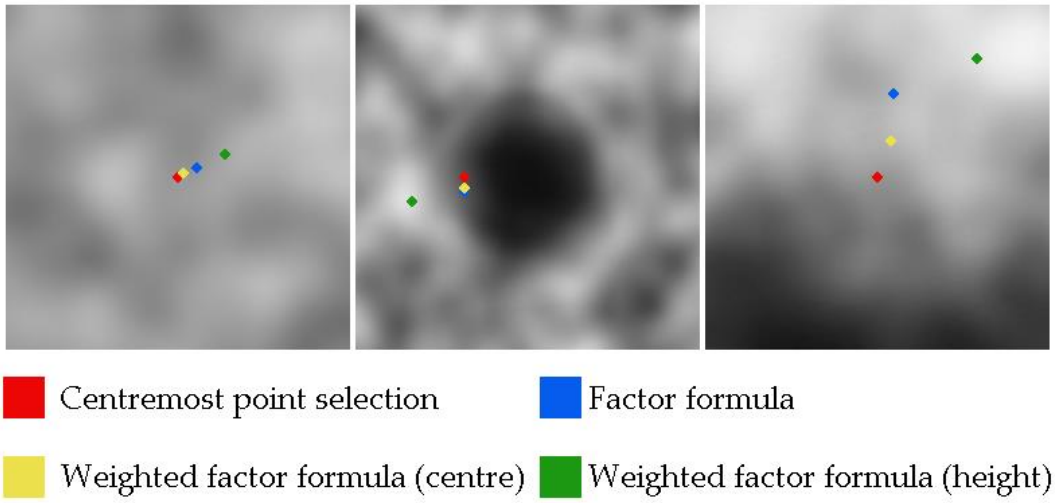

Figure 21: The four city citing methods upon the three heightmaps.

	Heightmap 1	Heightmap 2	Heightmap 3
Method 1: Centremost point selection	128, 128	81, 128	128, 128
Method 2: Factor formula	141, 121	81, 139	140, 65
Method 3: Weighted factors (centre)	132, 124	80, 136	137, 100
Method 4: Weighted factors (height)	163, 111	42, 146	201, 40
Mean coordinates	141, 121	73, 137.25	151.5, 83.25

Table 3: The coordinates of the centremost city points upon the three heightmaps.

In order to retrieve an accurate reading of the measure of dispersion between coordinates, a method of calculating the variance had to be devised. Various measures were considered, such as the Standard Distance Deviation or the Mahalonobis Distance, but due to the low number of sampled points involved these methods were not considered appropriate. We therefore decided to simply measure the Euclidian distance from each coordinate to the mean point upon each heightmap, and calculate a dispersion value by finding the average of these distances.

Put mathematically, we would say that the level of dispersion can be written as:

$$\frac{1}{n}\sum_{i=1}^{n}\|\boldsymbol{z}_i - \boldsymbol{c}\|$$

Where $\boldsymbol{z}_i$ is defined as every measured coordinate (i.e. $\boldsymbol{z}_i = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}$), and $\boldsymbol{c}$ is defined as the central or mean point. Calculating the distance from each point to the central point is a simple matter of gathering the Euclidian distance through use of the Pythagorean Theorem.

$$\|\boldsymbol{z}_i - \boldsymbol{c}\| = \sqrt{(x_i - c_1)^2 + (y_i - c_2)^2}$$

The results of these measurements can be seen in Table 4.

	Heightmap 1	Heightmap 2	Heightmap 3
Method 1: Centremost point selection	14.76	13.62	50.55
Method 2: Factor formula	0	10.15	21.57
Method 3: Weighted factors (centre)	9.49	9.09	22.15
Method 4: Weighted factors (height)	24.17	30.29	65.73
Mean distance (level of dispersion)	12.11	15.79	40

Table 4: The distance from each sited location to the mean point upon each heightmap, and the calculated level of dispersion.

4.2 CITY LAYOUT

It was decided that, in order to receive an accurate picture of the generated city's layout, renderings had to be made that clearly demonstrate the distribution of residential, governmental, and religious buildings. Traditional screenshots were deemed unsuitable for this analysis, as discerning the various types of buildings from an overhead view was considered to be difficult. Therefore, specialised renderings had to be created.

The rendering process was as follows. First, all extraneous models were removed. This involved the removal of the skybox, ground, water, road, and trees from the rendering pipeline. The buildings were stripped of their textures, and instead were assigned a coloured material according to their designated building type. The colours red, blue, and green were used to designate residential, governmental, and religious buildings respectively, and black was assigned to any remaining structures. Directional lighting was turned off, and the camera was adjusted to a top-down, orthographic perspective, allowing for a clear bird's eye view of the scene.

For consistency, a set of constants were put in place. The same flat terrain was used for each rendering. The city size was kept consistent (a CitySize value of 200, indicating a city circumference of approximately 3770 meters). The same set of key structures – namely, the forum, central temple, and amphitheatre – was used in all the renderings. Where randomness was required, we generated time-seeded pseudo-random numbers, as explained in section 3.2.4.1.

With these parameters in place, three renderings were produced, utilising the three proposed methods described section 3.2.4, Building Locations. That is, the first rendering demonstrates the use of a simple random number generator to stochastically assign the three types of generic building types to allotments. The second rendering demonstrates the use of sectioned districts to create "groups" of similarly structured buildings. The third rendering demonstrates the use of probability distribution in order to create an organic, semi-random set of districts.

The three renderings were combined into a single image and labelled in Figure 19.

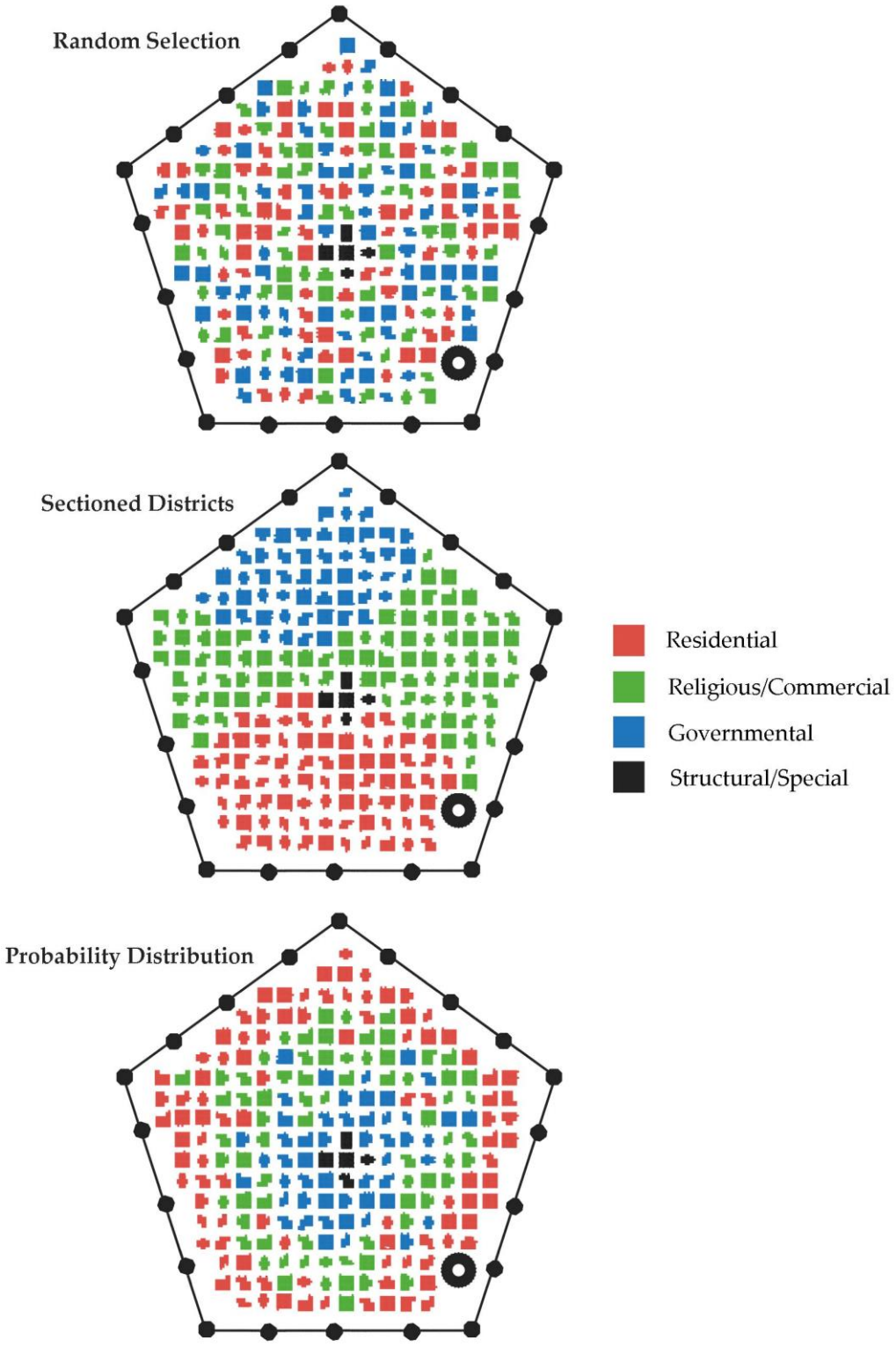

Figure 22: The three special renderings that demonstrate the building layout of a generated city.

Figure 23: A top-down rendering of a similarly sized city, without any of the special rendering effects applied.

Figure 24: An overhead, regular rendering of the city.

4.3 BUILDING GENERATION

In order to gauge the historical accuracy of the procedurally generated architecture in a meaningful manner, we produced rendered images of the models which could then be compared to illustrators' interpretations of Vitruvius' writings, and to architectural drawings and photographs of comparable real-life structures. This would allow for an assessment of the degree to which the generated architecture adhered to Vitruvius' measurements whilst also allowing us to assess the degree to which the digital models deviate from real life.

Out of all the buildings described in section 3.2.5, Building Generation, the temple was selected to be the primary focus of our comparison. There were a number of reasons that facilitated this decision.

The first was that Vitruvius paid a disproportionately large amount of attention to temples, dedicating two whole books to the dimensions of the structures. This level of attention was reflected in our implementation by way of a particularly detailed analysis of the structures (section 3.2.5.2), and it was decided that this standard ought to be maintained through the results stage for the purpose of consistency.

The second reason was that, as a result of modern preservation and reconstruction efforts, many Roman temples remain relatively unchanged, at least superficially so, from their architectural state at the time of construction. This made our attempts to obtain adequate photographs of the structures fairly straight-forward. By contrast, the number of other Roman buildings that have been preserved in a near-pristine state is relatively small, and as such we assessed that attempting to procure measurable photographs of these structures could prove to be an issue.

An additional consideration was that we believed Roman temples to be a more consistent representation of Vitruvian ideals than other architectural structures. A residential building may be subject to stylistic deviations at the hands of architects and builders, for example, and therefore may risk not conforming to the Vitruvian ideal. However, this is a speculative notion that deserves further study.

Renderings were made of the modelled temple from various angles in both an orthographic viewpoint, and in a more natural perspective (Figure 22). We also produced renderings of the other buildings described through our shape grammars (Figure 23).

For technical illustrations of the temple architecture described in De Architectura, various translations and editions were consulted. A version illustrated under the direction of Herbert Langford Warren (Vitruvius & Morgan, 1914) was selected to be

the primary source of comparison images, due to the book's comprehensive collection of clean, well-annotated technical diagrams (Figure 24). For the purpose of a close and meaningful comparison, we selected images from the book that conform to our choice of temple structure (i.e. single front-facing portico, tetrastyle layout, Doric order) wherever possible.

When selecting real-life examples of Vitruvian architecture for the purpose of comparison, we deliberately sought out temples that were in a relatively complete and undamaged condition

The first temple selected was the Maison Carrée in Nîmes, France, a structure notable for being a particularly well-preserved example of Vitruvian architecture (Jones, 2000). Originally built in 16 BC, the structure has been reconstructed and restored several times. It features six Corinthian columns across its portico, and twenty columns embedded in the cella walls. The temple measures 26.42 meters by 13.54 meters

The second temple to be selected was the Temple of Portunus, also known as the Temple of Fortuna Virilis, in Rome, Italy. It is notably smaller than the Maison Carrée, with a podium only measuring 19 meters by 10.5 meters. It features four Ionic columns across its portico.

Other Vitruvian temples were considered for the purpose of comparison, such as the harbour temple at Ulpia Traiana or the temple of Jupiter Stator at Rome, but the structures were deemed inappropriate for comparison due to their state of decay or ruin.

For both the Maison Carrée and the Temple of Portunus, photographs were drawn from the Wikimedia Commons depository (Figure 25, Figure 27), and architectural illustrations were drawn from Fletcher's A History of Architecture on the Comparative Method (Figure 26, Figure 28). These image sources were selected due to their visual clarity and technical accuracy.

With the image sources selected for both our rendered temple and for the comparison examples, we were able to make a set of measurements. First, the most immediately observable features of the assessed temple structures were documented: the style of the portico colonnade and the order of the columns. Next, by drawing from direct measurements of the necessary diagrams, sets of comparative ratios were measured with regards to the dimensions of the podium, columns, and cella for each temple.

Consistency was ensured by maintaining the same standards and restrictions for all measures. For example, we measured the podium dimensions from and to the very

ends of the architectural elements, bannisters included, and we measured the cella dimensions along the outer perimeter of the cella wall. The full list of observed measurements can be found in Table 5.

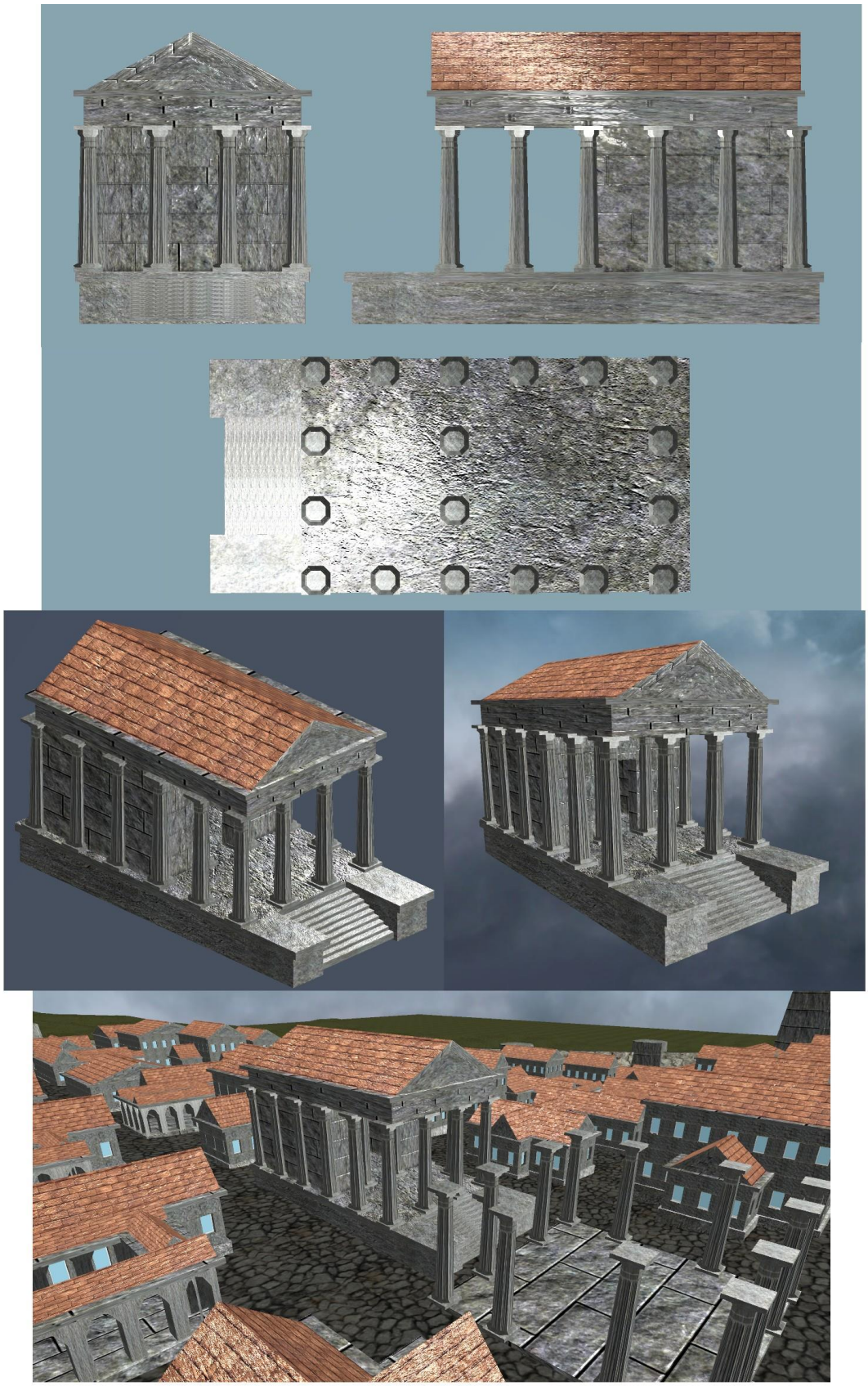

Figure 25: Four orthographic and two perspective renderings of the Vitruvian temple.

Figure 26: Renderings of an amphitheatre, a villa, and a governmental building.

These images have been removed

Figure 27: A compilation of Warren's various technical diagrams (Vitruvius & Morgan, 1914).

Figure 28: Photograph of Maison Carrée from Wikimedia Commons.

This image has been removed

Figure 29: Architectural illustration of Maison Carrée (Fletcher, 1921).

Figure 30: Photograph of the Temple of Portunus from Wikipedia Commons.

This image has been removed

Figure 31: Architectural illustration of the Temple of Portunus (Fletcher, 1921).

	Rendered temple	Warren's Vitruvian illustrations	Maison Carrée	Temple of Portunus
Portico colonnade style	Tetrastyle	Tetrastyle/ Hexastyle	Hexastyle	Tetrastyle
Column style	Doric	Doric/Ionic	Corinthian	Ionic
Podium width to length ratio	1:2.02	1:2.00	1:1.95	1:1.81
Temple height to podium width ratio (roof excluded)	1:1.03	1:1.21	1:1.11	1:1.06
Cella length to podium length ratio	1:1.99	1:1.61	1:1.99	1:1.98
Cella doorway height to cella height ratio	1:1.36	1:1.29	1:1.26	1:1.26
Column shaft height to full height ratio	1:1.12	1:1.07	1:1.25	1:1.13
Column diameter to shaft height ratio	1:6.33	1:6.30	1:7.38	1:7.65

Table 5: Comparison of the features and measurements of the various Vitruvian temples.

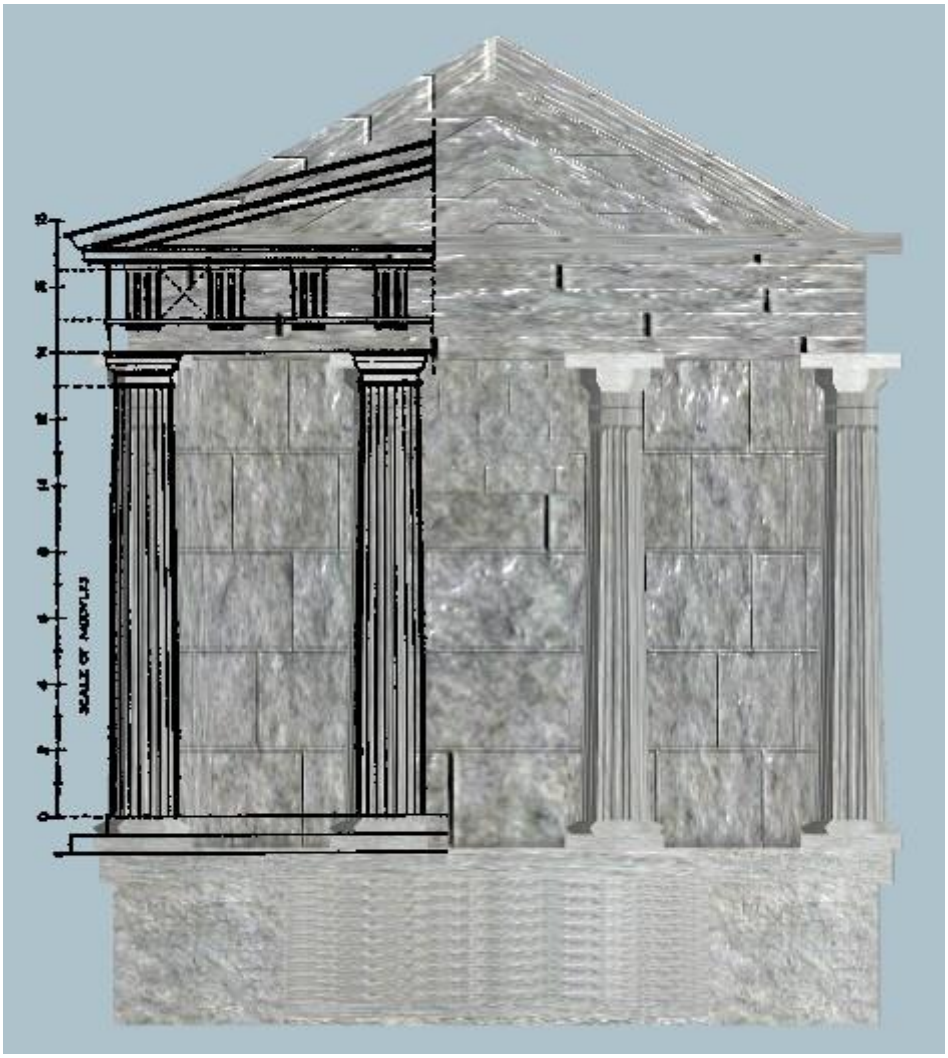

Figure 32: Overlay comparison of Warren's drawings with the rendered output.

4.4 Application Efficiency and Limitations

For the purpose of ensuring that the application is usable in real-time across a range of scenarios, the number of buildings, the number of polygons, and the average framerate of the application were measured across a range of city sizes.

To ensure consistent and meaningful results, a number of constants were put in place. The same terrain was used throughout. To minimize frame rate fluctuations, the frame rate was measured over a period of five seconds, and the mean result was calculated. To avoid discrepancies caused by a moving viewpoint, the camera was placed in the same fixed position overlooking the city across all test runs.

Despite these measures, it ought to be noted that there is still a degree of variation in the performance results due to the stochastic nature of the building generation process. We considered limiting the application to the production of a single type of building during the generation of the city, but such a measure would not accurately reflect the nature of the application.

"CitySize" was an arbitrary value that was used within the application for the purpose of calculating the city's radius. An approximate measure of 1 CitySize unit to 3 meters can be drawn, based upon the previously calculated measurements of road and insula dimensions (section 3.2.3).

The test was performed on a Toshiba laptop with a 2.3GHz processor, 6 GB of RAM, and a GeForce graphics card.

CitySize	City Radius (Meters)	Number of Buildings	Number of Polygons (triangles)	FPS (Frames Per Second)
100	300	99	87970	208
125	375	104	93610	201
150	450	110	101952	192
175	525	144	138146	185
200	600	197	173660	176
225	675	248	217226	164
250	750	293	257074	150

Table 6: The measured building count, polygon count, and framerate measurement at regularly positioned CitySize intervals.

The largest city that could be generated by the application had a CitySize value of 350, representing a city radius of 1050 meters and generating around 440 buildings. At larger values, the application stalled and crashed. We believe that this crash originates from a stack overflow in one of the arrays used for the purpose of storing building vertex data.

The smallest possible city that could be generated had a CitySize value of 1, although this resulted in clear object overlapping and graphical errors. The smallest realistic city that could be generated without causing graphical errors had a CitySize value of approximately 50, representing a city with a radius of 150 meters.

From archaeological record, we know that newly-founded Roman cities had a circumference of approximately 2 to 4 kilometres. This equates to a radius of 318 to 637 meters, or a CitySize value of approximately 106 to 212.3 units, well within the application boundary limits of 50 to 350.

5 CONCLUSION AND EVALUATION

5.1 CONCLUSIONS

The underlying purpose of this project was to provide a set of novel methods capable of converting architectural elements into sets of formal rules, which could then be adapted into a procedurally generated digital city model. Vitruvius' De Architectura was chosen as the exemplary test case. Section 0, Implementation, detailed the methodology used to formulate the various rules and grammars. A particular focus was placed on the use of a unique formal grammar for the purpose of describing the architectural shapes themselves.

In section 4, Results, the output of the application was recorded and measured within four subsections. For the purposes of clarity and consistency, the same subsections have been adopted in the following conclusion section; the output data is broken down section by section and analysed, and appropriate conclusions are drawn.

5.1.1 CITY POSITION

In section 3.2.1, various methods of positioning a city upon a basic landscape were discussed, and an adjustable formula was devised. In section 4.1, four variations of the formula were implemented across three heightmaps, providing a set of coordinates that demonstrated how the Vitruvian city could be positioned in a range of scenarios. A visual rendering of the siting methods upon each of the heightmaps can be seen in Figure 18.

With the sited points measured, the variance for each map was calculated by measuring the average Euclidian distance from the mean point to each of the sited city location's coordinates (Table 4).

We can draw observations about each siting method from the visual data in Figure 18, and from the coordinate data in Table 3.

Method 1, the centremost point selection method, sited cities in the very centre of the 256 by 256 map for two out of the three heightmaps. The exception, heightmap 2, featured an inaccessible centre area, and consequently the method chose a point adjacent to this area.

Method 2, the unweighted factor formula, and method 3, the factor formula weighted to the map's centre, produced fairly similar results. The two methods deviated most on heightmap 3, where the high point towards the northeast of the map caused method 2's sited location to weigh further in that direction.

Method 4, the factor formula weighted to height, appeared to produce sited locations near to the maps' highest points, with comparative disregard of the maps' centremost points. This is most apparent on heightmap 3, where method 4's siting location lies quite close to the map's northeast border.

We can also draw several conclusions from the data presented on Table 4. The most immediately observable facts are that heightmap 3, the landscape that consisted of a large slope, featured the greatest level of dispersion (an average distance from centre of 40 units), and that heightmap 1, the relatively flat surface, featured the smallest level of dispersion (an average distance from centre of 12.11 units). These results were likely brought about due to the discrepancy in the weighting of the height variable. Methods 2 and 4 in particular tended to have sited locations that skewed towards height, causing large levels of divergence on the third heightmap.

The implication of these results is that, under the chosen method of implementation, a flatter landscape produces more consistent results, regardless of which siting method is adopted. If we were to use a landscape that were entirely flat, all four of the city siting methods tested would have sited locations at the very centre of the map, providing an average dispersion level of zero. Conversely, terrain that is inconsistently mountainous is likely to yield a greater range of results, and so the design of the citing formula becomes of greater importance.

From these results, there appears to be no doubt that the weighted factor formula is the "best" of the three proposed city positioning formulae. The optional weighting allows for a large amount of variety, to the point where all reasonable user needs can be accounted for. Due to the scope of this project, we are unable to perform an assessment of what weighting would produce results closest to the historical record, but this is certainly something that could be explored through further studies.

5.1.2 CITY LAYOUT

Section 3.2.4, Building Locations, describes the methodology used for assigning the key structures of a Vitruvian city, and then outlines three potential methods for designating the remaining 'generic' structures. Section 4.2 demonstrates the results of this methodology by providing a set of three specialised renderings.

From the rendered images in Figure 19 and Figure 20, we can immediately draw an observation: regardless of the building location method used, certain structures remain consistently placed. The forum and its adjacent buildings – indicated on the specialised renderings by the five central black buildings – are consistently assigned to the same grid squares towards the centre of the city. The amphitheatre – the black ring – consistently lies towards the outside of the city, but within the confines of the walls. These results are in keeping with the methods used to assign the key structures, as described in the introductory paragraphs of section 3.2.4. This is also in keeping with the locations suggested by Vitruvius (De Architectura I, 7; De Architectura V).

We can also observe the differences in the building dispersion between the three rendered methods of Figure 19. The first rendering, depicting the random method described in section 3.2.4.1, is visibly lacking in order; any apparent patterns of grouping can be attributed to the nature of randomness. The second rendering, depicting the sectioned districts of section 3.2.4.2, shows the three types of building types allocated to three equally-sized districts. The third rendering, depicting the probability distribution method described in section 3.2.4.3, follows the predicted pattern of having mostly governmental buildings towards the centre, and mostly residential buildings towards the outer edge. Exceptions can be noted, such as the two outlying governmental buildings towards the top-right of the image, but again, this can be attributed to the nature of randomness.

It would be simplistic to say that the probability distribution method produces the "best" results, as there are likely to be cases in which the other two proposed methods produce results more befitting of a user's needs. However, for the purposes of our Vitruvian city scenario, the probability distribution method appeared to produce results closest to what we may expect a Roman city to look like. By that measure, the probability distribution method of building placement can be considered the most apt for our framework.

Also of note are the dimensions of the city wall. In section 3.2.2, we asserted that the pentagonal wall surrounding a Roman city ought to have a perimeter between approximately 1869 and 3744 metres. We also asserted that the spacing between the towers upon a city wall ought to be by approximately 182 meters by Vitruvius' measurements, but closer to 60 meters when going by historical record.

Using the conversion rate calculated in section 4.4, we can assert that the city in the rendered example has a circumference of approximately 3770 metres. From this, we can infer that the rendered pentagonal wall has a perimeter of 3527 metres, within the typical measurements gathered from historical record. Since we can count 20 towers upon the city wall in the rendered examples, each positioned regularly apart,

we can calculate the distance from any tower centre point to an adjacent tower centre point to be approximately 176 meters.

We believe this measurement to be as close to the Vitruvian ideal as possible within the given outer wall dimensions. If an additional tower were positioned on each side of the pentagonal wall, totalling 25 towers, then there would be a gap of approximately 141 meters between towers. If there were one fewer towers on each side, totalling 15 towers, then the gap would be approximately 235 meters. Consequently, although there is a small deviation between the current measurement and Vitruvius' recommendation, this is the smallest possible deviation attainable without altering the dimensions of the outer wall. As such, the implemented method of situating the outer wall's towers functions exactly as intended.

Despite this, a possible limitation can be observed when the city is viewed from overhead, as with Figure 20. It is immediately apparent that the city blocks, as defined in section 3.2.3, are homogenous squares, each only consisting of a small number of architectural structures. Without further studies, it is difficult to assess whether this peculiarity is a result of an error in the implementation, or whether it derives from a misinterpretation of Vitruvius' rules, or whether the application is functioning exactly as intended and this is how Vitruvius intended for an ideal city to be designed.

5.1.3 BUILDING GENERATION

Section 4.3 contains large amounts of rendered output, demonstrating the architecture produced by the grammars outlined in section 3.2.5. A large number of rendered images of the generated Vitruvian temple are included, along with several images of other notable pieces of generated architecture. Additionally, the rendered images of the temple are compared to diagrams of the Temple of Portunus, Maison Carrée, and an illustrator's interpretation of Vitruvius' rules, allowing us to make an analysis of the success of the rendered architecture itself.

Table 5 consists of a set of measurements, detailing the relative ratios of various architectural elements on each of the four tested temples. We will address the gathered data on a row-by-row basis.

First, we made note of the column style and portico layout for each temple. Our temple, as described in section 3.2.5.2, can be described as tetrastyle and Doric. This contrasts with Maison Carrée and the Temple of Portunus, which can be described as hexastyle and Corinthian, and tetrastyle and Ionic respectively. Since Warren's

illustrations are based on the descriptions in De Architectura, the illustrations cover a variety of temple and column styles, so this was noted accordingly.

Next, the podium width to length ratio was measured. Vitruvius recommended that a temple ought to have a length twice its width (De Architectura IV, 4), and Warren's illustrations proved to be perfect in this regard. The other measured temples, including the rendered temple, also delivered values very close to the one-to-two ratio.

The temple height to podium width ratio provides a measure for how 'squashed' a temple appears, when viewed from the front. The rendered temple provided the smallest ratio, indicating that it was the 'thinnest' of the measured temples, albeit only a little more so than the Temple of Portunus. Warren's illustrated temples proved to be the widest, although it ought to be noted that we received varying ratios depending on which of Warren's illustrations was used; the illustrator appeared to draw podiums of varying heights and thicknesses, depending on the context of the drawing.

The cella length to podium length ratio provides a measure of exactly how much of the podium is covered up by the cella chamber. The rendered temple, Maison Carrée, and the Temple of Portunus provided nearly identical results in this measure, all indicating that they had a podium nearly twice the length of the cella. Warren's illustrations deviated significantly. We believe this to be because Warren tended to exclude or shorten the illustrated stairway and bannisters when unnecessary, creating illustrated temples that could appear too short when measured out of context.

The cella doorway height to cella height demonstrates the relative height of the cella entranceway. The two real-life temples provided identical ratios for this measure, and the value gathered from Warren's illustrations was also quite similar. However, the rendered temples' ratio does not match the others. We believe that the source of this discrepancy stems from an erroneous interpretation of Vitruvius' writings. In the cella subsection of section 3.2.5.2, we asserted that the ratio of the cella doorway height to the entire cella height ought to be approximately 2.5 parts to 3.5 parts, giving a cella doorway height to cella height ratio of 1:1.4. This value was derived from a misunderstanding of which architectural parts were included in Vitruvius' measurements. With the proper elements included, we can expect to reach a value much closer to the 1:1.26 ratio produced by the real-life temples. Amusingly, we are not the first readers of Vitruvius to make this mistake; one scholar made note of having difficulty deciphering exactly what was included in Vitruvius' 2.5 to 3.5 ratio (Brenders, 2014).

The column shaft height to full column height ratio indicates how much of the column is taken up by the base and capital. The temples all offered fairly different values for this measure, but this was to be expected since three types of column orders were involved. However, there is a small discrepancy between the rendered temple and Warren's illustrations, which should not be the case since the two are of the same Doric order. We attribute this discrepancy to our inclusion of the square column base, which was described as optional by Vitruvius. If it is excluded, we would expect the discrepancy in the ratio between the two to become negligible.

The column diameter to shaft height ratio gives an approximate value of a column's relative thickness. Again, the measured ratios differ due to the different types of columns measured. As one would expect, the Ionic and Corinthian columns of the Temple of Portunus and Maison Carrée had larger ratios, indicating skinnier shafts than the stockier Doric columns of the rendered temple and Warren's illustrations. The two Doric temples gave very similar measurements, indicating that the rendered temple's Doric columns are at least accurate in this regard.

Put together, a picture is formed of the overall accuracy of the rendered temple, and by extension the accuracy of our proposed shape grammar syntax. Discrepancies exist between the measurements derived from our renderings and the measurements drawn from Warren's drawings, but many of these discrepancies are small enough to be dismissed as negligible mistakes made during the measuring process, or differences stemming from subtle ornate details in the measured images.

The overlayed comparison of one Warren's illustrations over the rendered output provides evidence of this (Figure 29); even though the silhouettes of the two images differ, there is a striking similarity between the two, and the significant architectural elements, such as the column dimensions and intercolumniation space, align perfectly. In this regard, the shape grammar must be considered to be successful.

5.1.4 *Application Efficiency and Limitations*

In section 4.4, measurements were provided of the number of buildings generated by the application across a set of city sizes, along with accompanying polygon counts and framerate readings. The application was demonstrated to be capable of producing cities within the measurements typically found in the historical record; a city radius between 150 and 1050 meters could be generated by the application, effectively encompassing the radius of a newly-founded Roman city which typically measured between 318 to 637 meters.

Also of note is the fact that the application proved capable of maintaining a high framerate, even when large numbers of polygons were on-screen. With 293 buildings totalling 257074 polygons in a single environment, the application maintained a framerate of 150 frames per second (Table 6). A high framerate is not considered to be the sole indicator that a simulation is capable of running in real-time, but given how the running application proved responsive to navigational commands under the most strenuous of tests, we assert that this application meets the criteria for a real-time simulation.

5.2 Discussion

In the Aims and Objectives section, we outlined a set of criteria for the purpose of ensuring that the designed formal grammar would be usable and appropriate. With the context of the implemented models and statistical results, we can make an assessment on the degree to which the criteria were met, and consequently we can judge the success of our overarching framework.

5.2.1 *First Objective*

The first objective was to provide a set of procedural methods, functions, or techniques for the purpose of describing the various elements of the urban environment.

This was accomplished to a high level of detail. A large portion of our implementation section described the process behind the choice of various formulae and techniques. Particular attention was paid to the positioning of the city upon a given heightmap, and the allocation of building allotments within the city

boundaries. The results of this, and the surrounding discussion of what methods were the most successful, were documented under the City Position and City Layout subsections.

5.2.2 SECOND OBJECTIVE

The second objective laid out a set of criteria that we aimed to adhere to in order to produce a usable and deterministic shape grammar syntax.

We proposed that the grammar must be specific and precise. This target has been met, as demonstrated through the results of Table 5. The images of our rendered temple and the illustrator's interpretation of Vitruvian architecture are remarkably similar. We identified discrepancies between the two, but these are likely to have stemmed from mistakes in the measuring process, not errors stemming from the shape grammar itself. For all practical intents and purposes, we discovered that the production rules written for the Vitruvian temple created rendered results that matched the prototypical structures with a desirable level of precision.

We proposed that the grammar must be flexible enough to encompass a wide variety of architecture. This criterion was effectively demonstrated through the range of architectural elements that were produced with the grammar rules. For example, the grammar proved capable of defining the curved surfaces of columns, the repeating nature of windows, and the semi-stochastic nature of building dimensions without difficulty. The grammar was not used to define certain details, such as the leaves upon a Corinthian capital, but we do not consider this to be a limitation of the grammar as much as a limitation on our time to implement such details. We did not observe any significant restrictions of the grammar's capabilities within the scope of this project.

We proposed that the grammar must be free of ambiguity in order to ensure that the output results are deterministic. This criterion was sufficiently met, as demonstrated by the fact that, where desired, implemented grammars were capable of returning the same model data on multiple runs of the application. Some may argue that the implementation of randomness in some of the architecture implies that some of the grammars are non-deterministic by definition, but this is not wholly accurate. There is a clear distinction between a formal grammar that is non-deterministic due to ambiguity arising from multiple unclarified derivations, and a formal grammar that is non-deterministic due to derivations being selected through a randomly generated number. We observed no ambiguities that arose from the grammar syntax itself, and consequently we consider our proposed grammar to be effectively deterministic.

We proposed that the grammar ought to be comprehensible on a human level. This criterion is open to some degree of subjectivity. Although we believe the system of superscript and subscript attachments to symbols to be legible once explained, it is undeniable that the grammar rules become harder to read and interpret as they grow in size. User tests would be one effective way of measuring whether or not the grammar syntax is comprehensible, both from the perspective of an industry professional, and from the perspective of a layperson.

With these criteria in mind, we believe the shape grammar syntax proposed and defined in section 3.2.5 to be novel, functional, and purposeful. Additionally, we believe the other procedural methods defined in this thesis to be integral to the success of the application, even if not wholly novel or noteworthy in their own right. Consequently, we consider the application to be a cohesive collection of successfully implemented grammars, capable of procedurally generating a digital Vitruvian city with a strong degree of accuracy and detail.

5.2.3 Third and Fourth Objectives

The third objective required the demonstrated use of our grammar framework to interpret the various architectural instructions written by Vitruvius. The fourth objective was to describe the process of converting the procedural rules and grammar framework into a digitally modelled city.

These objectives can perhaps both be considered fulfilled through the documented implementation itself. Section 3.2.5 extensively details the process by which descriptive rules were adapted into formal grammar rulesets. Section 7.2, BuildCity.cpp, stands as a demonstration of the way in which these rules were adapted into programmed code.

We began this thesis by asking whether a framework could be designed for the purpose of converting a set of historical, architectural descriptions into a digital, modelled format. It would be presumptuous to consider this thesis a unified take on the question, especially considering the vastness and diversity of architectural writings available. However, we believe that the subject has been explored to a degree that we feel comfortable asserting that we have developed such a framework successfully.

5.3 PROJECT LIMITATIONS

Although many of the sections on architecture were covered in depth, some areas of Vitruvius' De Architectura were not touched upon within the scope of the project.

In some cases, the material was deemed irrelevant because it had little to do with architecture. For example, book IX of De Architectura covers a variety of subjects, from wrestling, to Pythagorean mathematics, to astronomy. Although the book is of academic significance, it only has a tenuous link to the construction of Roman cities, and as such the material had no place in the scope of this project.

Other sections of De Architectura were relevant to the subject of architecture, but ignored due to their specificity or detail. These sections would require a significant degree of attention and time to be thoroughly and accurately implemented. For example, in book VI Vitruvius details the considerations that must be made for the choice of building materials in the construction of a house, with regards to the local climate and culture. This is entirely related to the subject of architecture, and the material described could theoretically be incorporated into a virtual building by way of selectively applied custom textures. However, designing and implementing functions for the purpose of generating or selecting textures would be a time-consuming task that could detract from the attention given to the architecture itself.

Section 4.4, Application Efficiency and Limitations, listed the limitations that were reached with the developed application itself. Most notably, the application was incapable of generating cities with a city radius above 1050 meters or below 150 meters. Although this was acceptable for the purpose of the application, it can certainly be considered a limitation of the program.

Some of the procedural methods implemented in this project were entirely appropriate within the context of Vitruvian cities, but would be of limited use if applied to other scenarios. For example, in section 3.2.3 a method of defining the gridded road system was described. This was a simple solution that was befitting of the specifications laid out by Vitruvius, and allowed for a certain degree of historical accuracy. However, the solution would be unsuitable if, for example, a user wished to recreate a city with a radial road system. This does not demonstrate a failure of the application, but it does indicate that some of the described methods have not been designed with flexibility or universal purpose in mind. In this regard, the system can be considered limited.

One other limitation of the project that we encountered is that of user fallibility. In section 5.1.3, we explained one situation in which a grammar production rule worked as intended, but the produced results were inconsistent with what was

desired. This was attributed to a misinterpretation of the writings on which the grammar rule was based. This does not demonstrate a limitation of the grammar system, but it can be viewed as a limitation of our chosen implementation pipeline; if a more automated or systematic method of inputting grammar rules were adopted, it is possible that the chance of an error arising would be minimised.

5.4 Project Contributions

Having established what aims have been fulfilled within the context of this thesis, we can directly address what we believe to have been contributed to the academic body of knowledge.

The largest contribution from this thesis is the overarching framework. This takes the form of a set of techniques and methodologies that were utilized in the process of adapting historical descriptions into digital models.

Perhaps the most significant part of this framework is the novel shape grammar. The grammar format, as described in section 3.2.5, is designed to be human-readable and writeable. As such, the rules pertaining to the various Roman structures can be appropriated, modified, and implemented into a separate modelling or rendering application with relative ease.

Additionally, the rules of the shape grammar have been clarified to the point where we believe that, with little effort, a person could describe all manner of architecture by following the methodology presented in this thesis. This may prove to be of particular use to historians wishing to document the shape of a structure in a deterministic fashion, or for architects wishing to describe a structure in a format that can be understood by a programmer versed in procedural generation.

Finally, we have contributed a digital model of a Vitruvian city. Although it would probably be inaccurate to say that this is the first digital model of a Vitruvian city, we feel confident asserting that this is the first to have been created wholly through procedural methods. The city model is in a format capable of being easily converted and read by a variety of applications and, as demonstrated through our application, it can be navigated on a pedestrian level. The model itself may be of academic significance to historians, architects, and other scholars wishing to see or utilise a 3D representation of Vitruvian architecture.

5.5 FUTURE WORK

The project has the potential to be expanded upon in several regards.

First, the study into the adaptation of Vitruvius' writing into procedural grammars could be furthered. Vitruvian structures that were not considered to be in the scope of this project, such as aqueducts and harbours, have the potential to be written with the designed shape grammar syntax, and then generated as digital models. Some areas, such as Vitruvius' description of columns, are detailed and complex enough to warrant entire further studies in themselves. With enough detail, an entire procedural map of Vitruvius' work could be created, which could prove to be of great historical and academic significance.

Second, the comparison of the rendered output to the underlying grammars and rules, as described in section 4, could be furthered. Out of all the generated architecture, only the Vitruvian temple was thoroughly analysed and compared. By providing thorough comparisons of the other pieces of architecture, a greater measurement of the application's overall accuracy could be drawn, and consequently the success of the project could be more thoroughly evaluated.

Finally, the formal grammar syntax described in section 3.2.5 has the potential to be refined and expanded upon, to the point where it could be considered a viable method of describing a variety of architectural structures within the fields of historic recreation, architectural design, and urban planning. Feedback from user testing would be fundamental in the identification of areas where the grammar syntax needs adjustment. Improvements could then be made to create a full grammar system that is robust and flexible enough to accommodate for every potential use case.

With enough improvements, the grammar syntax may even be versatile enough to describe all manner of two and three dimensional geometric shapes in a clear and unambiguous way, not just architectural forms. Such a shape grammar would not be the first of its kind, but we believe it would be novel in its design, format, and underlying purpose. The grammar's unique attributes would ensure that it fulfils a necessary and currently unfilled niche in the field of procedural generation.

6 BIBLIOGRAPHY

Addison, P. S., 1997. *Fractals and Chaos - An Illustrated Course.* s.l.:Institute of Physics Publishing.

Alexander, C. et al., 1977. A Pattern Language. *Oxford University Press, New York.*

Brenders, F., 2014. *Vitruvius: De architectura Libri X.* [Online]
Available at: http://www.vitruvius.be/boek4h6.htm
[Accessed July 2014].

Bullmore, E. T. et al., 1999. Global, voxel, and cluster tests, by theory and permutation, for a difference between two groups of structural MR images of the brain. *Medical Imaging,* 18(1), pp. 32-42.

Cheung, G., Kanade, T., Bouguet, J.-Y. & Holler, M., 2000. A real time system for robust 3D voxel reconstruction of human motions. *Computer Vision and Pattern Recognition,* Volume 2, pp. 714-720.

Cui, J., Chow, Y. W. & Zhang, M., 2011. A Voxel-based Octree Construction Approach for Procedural Cave Generation. *IJCSNS International Journal of Computer Science and Network Security,* 11(6), pp. 160-168.

De Malsche, K., De Boodt, J., Joos, E. & Ongena, O., 1983. *Mathematical Properties in Ancient Theatres and Amphitheatres.* [Online]
Available at: http://mathsforeurope.digibel.be/amphi.htm
[Accessed 2 March 2014].

Downing, F. & Flemming, U., 1981. The Bungalows of Buffalo. *Environment and Planning B.*

Flemming, U., 1987. More than the Sum of its Parts: the Grammar of Queen Anne Houses. *Environment and Planning.*

Fletcher, B., 1921. *A History of Architecture on the Comparative Method.* Sixth edition ed. New York: Charles Scribner's Sons.

Fournier, A., Fussell, D. & Carpenter, L., 1982. Computer Rendering of Stochastic Models. *ACM Siggraph.*

Goncalves, A., Magalhaes, L., Moura, J. & Chalmers, A., 2009. High Dynamic Range - A Gateway for Predictive Ancient Lighting. *Journal on Computing and Cultural Heritage,* 2(1).

Greuter, S., Parker, J., Stewart, N. & Leach, G., 2003. Real-time Procedural Generation of 'Pseudo Infinite' Cities. *Graphite,* pp. 87-95.

Greuter, S., Stewart, N., Parker, J. & Leach, G., 2003. Undiscovered Worlds – Towards a Framework for Real-Time Procedural World Generation. *MelbourneDAC.*

Groenewegen, S., Smelik, R. M., Kraker, K. J. d. & Bidarra, R., 2009. Procedural City Layout Generation Based on Urban Land Use Models. *Proceedings of Eurographics.*

Gutierrez, D., Sundstedt, V., Gomez, F. & Chalmers, A., 2008. Modeling Light Scattering for Virtual Heritage. *Journal on Computing and Cultural Heritage,* 1(2).

Hausdorff, F., 1972. *Dictionary of Scientific Biography.*

Hebbert, M. & Jankovic, V., 2009 . Street Canyons and Canyon Streets: the strangely separate histories of urban climatology and urban design. *Climate Science in Urban Design.*

Huang, J. et al., 2009. *An Evaluation of Shape/Split Grammars for Architecture,* s.l.: University of Waterloo.

Jones, M. W., 2000. *Principles of Roman Architecture.* s.l.:Yale University Press.

Kelly, G. & McCabe, H., 2006. A Survey of Procedural Techniques for City Generation. *ITB Journal,* Volume 14, pp. 87-130.

Kim, Z., Huertas, A. & Nevatia, R., 2000. Automatic Description of Complex Buildings with Multiple Images. *5th IEEE Workshop on Applications of Computer Vision.*

Koch, H. V., 1904. Sur une courbe continue sans tangente obtenue par une construction géométrique élémentaire.

Laycock, R. G. & Day, A. M., 2003. Automatically Generating Large Urban Environments Based on the Footprint Data of Buildings. *Proceedings of the ACM Symposium on Solid and Physical Modeling.*

Laycock, R. G., Drinkwater, D. & Day, A. M., 2008. Exploring Cultural Heritage Sites Through Space and Time. *ACM Journal on Computing and Cultural Heritage,* 1(2).

Lechner, T. et al., 2004. *Procedural Modeling of Land Use in Cities,* Illinois: Northwestern University.

Lesmoir-Gordon, N., Rood, W. & Edney, R., 2000. *Introducing Fractal Geometry.* s.l.:Cambridge.

Lindenmayer, A., 1968. Mathematical models for Cellular Interaction in Development.

Lluch, J., Camahort, E. & Vivo, R., 2003. Procedural Multiresolution for Plant and Tree Rendering. *AFRIGRAPH.*

Lynch, K., 1960. *The Image of the City.* Camebridge: MIT Press.

Mandelbrot, B., 1967. How Long is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension. *Science.*

Mandelbrot, B. B., 1982. *The Fractal Geometry of Nature.* s.l.:W.H. Freeman & Co..

Mateo-Babiano, I. & Ieda, H., 2005. Street Space Renaissance: A Spatio-Historical Survey of two Asian Cities. *Journal of the Eastern Asia Society for Transportation Studies,* Volume 6.

Miller, G. S. P., 1986. The Definition and Rendering of Terrain Maps. *ACM Siggraph.*

Mueller, P. et al., 2006. Procedural Modeling of Buildings. *ACM Transactions on Graphics.*

Muller, P., Vereenooghe, T., Ulmer, A. & Van Gool, L., 2005. Automatic reconstruction of roman housing architecture. *Recording, Modeling and Visualization of Cultural Heritage.*

Musgrave, F. K., Kolb, C. E. & Mace, R. S., 1989. The synthesis and rendering of eroded fractal terrains.. *ACM SIGGRAPH Computer Graphics,* 23(3), pp. 41-50.

Oppenheimer, P. E., 1986. Real time design and animation of fractal plants and trees. *SIGGRAPH '86 Proceedings of the 13th annual conference on Computer graphics and interactive techniques ,* 20(4), pp. 55-64.

Paden, R., 2001. Values and Planning: The Argument from Renaissance Utopianism. *Ethics, Place & Environment: A Journal of Philosophy & Geography,* 4(1).

Parish, Y. I. H. & Muller, P., 2001. Procedural Modeling of Cities. *ACM Siggraph.*

Perlin, K., 1985. An Image Synthesizer. *ACM Siggraph Computer Graphics,* 19(3), pp. 287-296.

Pickover, C. A., 2009. *The Math Book: From Pythagoras to the 57th Dimension, 250 Milestones in the History of Mathematics.* s.l.:Sterling Publishing Company Inc.

Pollefeys, M. et al., 2003. Visual Modeling with a Hand-Held Camera. *International Journal on Computer Vision,* 54(3).

Prusinkiewicz, P. & LindenMayer, A., 1990. The Algorithmic Beauty of Plants. *Springer-Verlag.*

Remondino, F., Girardi, S., Rizzi, A. & Gonzo, L., 2009. 3D Modeling of Complex and Detailed Cultural Heritage Using Multi-Resolution Data. *Journal on Computing and Cultural Heritage,* 2(1).

Rhoades, J. et al., 1992. Real-Time Procedural Textures. *I3D Proceedings of the 1992 symposium on Interactive 3D graphics.*

Romano, D. G., 2003. *City Planning, Centuriation, and Land Division in Roman Corinth: Colonia Laus Iulia Corinthiensis & Colonia Iulia Flavia Augusta Corinthiensis.,* s.l.: American School of Classical Studies at Athens.

Schpok, J., Simons, J., Ebert, D. S. & Hansen, C., 2003. A real-time cloud modeling, rendering, and animation system. *ACM Siggraph/Eurographics,* pp. 160-166.

Seitz, S. & Dyer, C., 1997. Photorealistic scene reconstruction by voxel coloring. *Computer Vision and Pattern Recognition,* pp. 1067-1073.

Stava, O. et al., 2010. Inverse Procedural Modeling by Automatic Generation of L-Systems. *Eurographics,* 29(2).

Stiny, G., 1975. Pictorial and Formal Apects of Shapes and Shape Grammars. *Birkhauser, Basel, Switzerland.*

Stiny, G., 1980. Introduction to shape and shape grammars. *Environment and Planning B,* Volume 7.

Susaki, J., 2013. Knowledge-Based Modeling of Buildings in Dense Urban Areas by Combining Airborne LiDAR Data and Aerial Images. *Remote Sensing,* 5(11), pp. 5944-5968.

Trochet, H., 2009. *A History of Fractal Geometry.* [Online]
Available at: http://www-history.mcs.st-and.ac.uk/HistTopics/fractals.html
[Accessed 21 08 2012].

Vanegas, C. A. et al., 2009. Modeling the Appearance and Behaviour of Urban Spaces. *Proceedings of Eurographics.*

Vegetius, ~390. *Epitoma Rei Militaris.* s.l.:s.n.

Vitruvius & Morgan, T. b. M. H., 1914. *The Ten Books on Architecture.* s.l.:Harvard University Press.

Vitruvius, n.d. *De Architectura.* s.l.:s.n.

Wonka, P., Wimmer, M., Sillion, F. & Ribarsky, W., 2003. Instant Architecture. *Siggraph.*

Yong, L., Mingmin, Z., Yunliang, J. & Haiying, Z., 2012. Improving Procedural Modeling with Semantics in Digital Architectural Heritage. *Computer and Graphics,* 36(3), pp. 178-184.

7 APPENDIX

Sections 7.1 and 7.2 are copies of the two most significant code files used in the development of the application: CityMain.cpp, the "main" file that handled the building allocation, road generation, and rendering process; and BuildCity.cpp, which contained the implemented shape grammars.

Sections 7.3 and 7.4 are copies of XFilePart1.txt and XFilePart2.txt, the two files that were used in process of exporting the virtual city data to a .x model file.

For a comprehensive copy of the code files used, along with the accompanying textures, assets, and a compiled version of the application, please see www.sketchylogic.com/romancity.

For a video of the application running, please see https://www.youtube.com/watch?v=OmZBYUujDyY

7.1 CITYMAIN.CPP

```
//Necessary includes
#include "d3dApp.h"
#include "DirectInput.h"
#include "BuildCity.h"
#include <crtdbg.h>
#include "GfxStats.h"
#include <list>
#include <ctime>
#include "Terrain.h"
#include "Camera.h"
#include "Water.h"
#include "Sky.h"
#include "Vertex.h"

using namespace std;

//Set up a prototype object structure that contains a mesh and the
necessary textures.
struct Object3D
{
	Object3D()
	{
		mesh = 0;
	}
	~Object3D()
	{
		ReleaseCOM(mesh);
		for(UINT i = 0; i < textures.size(); ++i)
			ReleaseCOM(textures[i]);
	}

	ID3DXMesh* mesh;
	std::vector<Mtrl> mtrls;
	std::vector<IDirect3DTexture9*> textures;
	AABB box;
};

//Set up the necessary build functions and D3D-specific rendering
objects.
class RomanCity : public D3DApp
{
public:
	RomanCity(HINSTANCE hInstance, std::string winCaption,
D3DDEVTYPE devType, DWORD requestedVP);
	~RomanCity();

	bool checkDeviceCaps();
	void onLostDevice();
	void onResetDevice();
	void updateScene(float dt);
	void drawScene();
	void drawObject(Object3D& obj, const D3DXMATRIX& toWorld);

	void buildTrees();
	void MakeAnXFile();
	void buildRoads();
	void buildCity();
```

```
	void buildFX();
private:
	GfxStats* mGfxStats;
	Terrain*  mTerrain;
	Water*    mWater;
	float mTime;

	Sky* mSky;
	ID3DXMesh* mSceneMesh;
	D3DXMATRIX mSceneWorld;
	D3DXMATRIX mSceneWorldInv;

	Object3D mRome;
	D3DXMATRIX mRomeWorld;
	Object3D mTrees[4];
	static const int NUM_TREES = 30;
	D3DXMATRIX mTreeWorlds[NUM_TREES];
	Object3D mBuildings[4];
	static const int NUM_BUILDS = 30;
	D3DXMATRIX mBuildWorlds[NUM_BUILDS];

	std::vector<Mtrl> mSceneMtrls;
	std::vector<IDirect3DTexture9*> mSceneTextures;

	IDirect3DTexture9* mSceneNormalMaps[7];

	IDirect3DTexture9* mWhiteTex;

	ID3DXEffect* mFX;
	D3DXHANDLE   mhTech;
	D3DXHANDLE   mhWVP;
	D3DXHANDLE   mhWorldInv;
	D3DXHANDLE   mhEyePosW;
	D3DXHANDLE   mhWorld;
	D3DXHANDLE   mhTex;
	D3DXHANDLE   mhMtrl;
	D3DXHANDLE   mhLight;
	D3DXHANDLE   mhNormalMap;

	DirLight mLight;

	bool mFreeCamera;
};


int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
                   PSTR cmdLine, int showCmd)
{
	#if defined(DEBUG) | defined(_DEBUG)
		_CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF |
_CRTDBG_LEAK_CHECK_DF );
	#endif

	srand(time(0));

	// Set up camera and viewpoint
	Camera camera;
	gCamera = &camera;

	RomanCity app(hInstance, "Roman City", D3DDEVTYPE_HAL,
D3DCREATE_HARDWARE_VERTEXPROCESSING);
```

```
	gd3dApp = &app;

	DirectInput di(DISCL_NONEXCLUSIVE|DISCL_FOREGROUND,
DISCL_NONEXCLUSIVE|DISCL_FOREGROUND);
	gDInput = &di;

    return gd3dApp->run();
}


RomanCity::RomanCity(HINSTANCE hInstance, std::string winCaption,
D3DDEVTYPE devType, DWORD requestedVP)
: D3DApp(hInstance, winCaption, devType, requestedVP)
{

	InitAllVertexDeclarations();

	mGfxStats = new GfxStats();

	SetCurrentDirectory("Art/");

	mFreeCamera = 0;

	//Set up skybox
	mSky = new Sky("newsky.dds", 10000.0f);

	//Set up terrain, with appropriate textures
	mTerrain = new Terrain(513, 513, 4.0f, 4.0f,
		"coastMountain5132.raw", "newgrass.jpg",  "dirt.dds",
		"cobble3.jpg", "blendnew.png", 0.65f, 0.0f);

	//Set up and move the water as appropriate
	D3DXMATRIX waterWorld;
	D3DXMatrixTranslation(&waterWorld, 0.0f, 50.0f, 0.0f);

	//Set up lighting
	mLight.dirW = D3DXVECTOR3(0.0f, -2.0f, -1.0f);
	D3DXVec3Normalize(&mLight.dirW, &mLight.dirW);

	mLight.ambient = D3DXCOLOR(0.4f, 0.4f, 0.4f, 1.0f);
	mLight.diffuse = D3DXCOLOR(0.8f, 0.8f, 0.8f, 1.0f);
	mLight.spec    = D3DXCOLOR(0.6f, 0.6f, 0.6f, 1.0f);

	Mtrl waterMtrl;
	waterMtrl.ambient   = D3DXCOLOR(0.26f, 0.23f, 0.3f, 0.90f);
	waterMtrl.diffuse   = D3DXCOLOR(0.26f, 0.23f, 0.3f, 0.90f);
	waterMtrl.spec      = 1.0f*WHITE;
	waterMtrl.specPower = 64.0f;

	Water::InitInfo waterInitInfo;
	waterInitInfo.dirLight = mLight;
	waterInitInfo.mtrl     = waterMtrl;
	waterInitInfo.vertRows         = 128;
	waterInitInfo.vertCols         = 128;
	waterInitInfo.dx               = 10.0f;
	waterInitInfo.dz               = 10.0f;
	waterInitInfo.waveMapFilename0 = "wave0.dds";
	waterInitInfo.waveMapFilename1 = "wave1.dds";
	waterInitInfo.waveMapVelocity0 = D3DXVECTOR2(0.05f, 0.08f);
	waterInitInfo.waveMapVelocity1 = D3DXVECTOR2(-0.02f, 0.1f);
	waterInitInfo.texScale = 32.0f;
```

```
	waterInitInfo.toWorld = waterWorld;

	mWater = new Water(waterInitInfo);
	mWater->setEnvMap(mSky->getEnvMap());

	//Call the functions to generate the appropriate tree, road,
and building models
	buildTrees();
	//buildRoads();
	buildCity();

	// Initialize camera.
	gCamera->pos().y = 3.0f;
	gCamera->pos().z = -10.0f;
	gCamera->setSpeed(10.0f);
	mGfxStats->addVertices(mSceneMesh->GetNumVertices());
	mGfxStats->addTriangles(mSceneMesh->GetNumFaces());
	mGfxStats->addVertices(mSky->getNumVertices());
	mGfxStats->addTriangles(mSky->getNumTriangles());

	buildFX();

	onResetDevice();
}

RomanCity::~RomanCity()
{
	delete mGfxStats;
	delete mSky;

	ReleaseCOM(mFX);

	ReleaseCOM(mSceneMesh);
	for(UINT i = 0; i < mSceneTextures.size(); ++i)
		ReleaseCOM(mSceneTextures[i]);

	ReleaseCOM(mWhiteTex);
	ReleaseCOM(mSceneNormalMaps[0]);
	ReleaseCOM(mSceneNormalMaps[1]);

	delete mTerrain;
	delete mWater;

	DestroyAllVertexDeclarations();
}

//Function that checks for proper shader support
bool RomanCity::checkDeviceCaps()
{
	D3DCAPS9 caps;
	HR(gd3dDevice->GetDeviceCaps(&caps));

	if( caps.VertexShaderVersion < D3DVS_VERSION(2, 0) )
		return false;

	if( caps.PixelShaderVersion < D3DPS_VERSION(2, 0) )
		return false;

	return true;
}
```

```
void RomanCity::onLostDevice()
{
	mGfxStats->onLostDevice();
	mTerrain->onLostDevice();
	mWater->onLostDevice();
	mSky->onLostDevice();
	HR(mFX->OnLostDevice());
}

void RomanCity::onResetDevice()
{
	mGfxStats->onResetDevice();
	mSky->onResetDevice();
	mTerrain->onResetDevice();
	mWater->onResetDevice();
	HR(mFX->OnResetDevice());

	float w = (float)md3dPP.BackBufferWidth;
	float h = (float)md3dPP.BackBufferHeight;
	gCamera->setLens(D3DX_PI * 0.25f, w/h, 1.0f, 5000.0f);
}

//Update loop - this is called every tick
void RomanCity::updateScene(float dt)
{
	//Update stats as necessary. This is used for calculating
framerates, poly counts etc.
	mTime += dt;
	mGfxStats->update(dt);
	gDInput->poll();

	//Check key inputs and change camera mode if necessary
	if( gDInput->keyDown(DIK_N) )
		mFreeCamera = false;
	if( gDInput->keyDown(DIK_M) )
		mFreeCamera = true;

	if( mFreeCamera )
	{
		gCamera->update(dt, 0, 0);
	}
	else
	{
		gCamera->update(dt, mTerrain, 2.3f);
	}

	static float time = 0.0f;
	time += dt;
	mLight.dirW.x =  2.0f;
	mLight.dirW.z =  3.0f;
	mLight.dirW.y =  -1.0f;

	mWater->update(dt);

	D3DXVec3Normalize(&mLight.dirW, &mLight.dirW);
}

void RomanCity::drawScene()
{
	HR(gd3dDevice->BeginScene());
```

```
	mSky->draw();

	HR(mFX->SetValue(mhLight, &mLight, sizeof(DirLight)));
	HR(mFX->SetMatrix(mhWVP, &(mSceneWorld*gCamera->viewProj())));
	HR(mFX->SetValue(mhEyePosW, &gCamera->pos(),
sizeof(D3DXVECTOR3)));

	UINT numPasses = 0;
	HR(mFX->Begin(&numPasses, 0));
	HR(mFX->BeginPass(0));

	for(UINT j = 0; j < mSceneMtrls.size(); ++j)
	{
		HR(mFX->SetValue(mhMtrl, &mSceneMtrls[j], sizeof(Mtrl)));

		// Apply textures when available. Otherwise, use plain
white.
		if(mSceneTextures[j] != 0)
		{
			HR(mFX->SetTexture(mhTex, mSceneTextures[j]));
		}

		else
		{
			HR(mFX->SetTexture(mhTex, mWhiteTex));
		}

		HR(mFX->SetTexture(mhNormalMap, mSceneNormalMaps[j]));

		HR(mFX->CommitChanges());
		HR(mSceneMesh->DrawSubset(j));
	}

	//Use alpha mask for trees
	HR(gd3dDevice->SetRenderState(D3DRS_ALPHATESTENABLE, true));
	HR(gd3dDevice->SetRenderState(D3DRS_ALPHAFUNC,
D3DCMP_GREATEREQUAL));
	HR(gd3dDevice->SetRenderState(D3DRS_ALPHAREF, 200));

	// Draw the trees: NUM_TREES/4 of each of the four types.
	for(int i = 0; i < NUM_TREES; ++i)
	{
		if( i < NUM_TREES/4 )
			drawObject(mTrees[0], mTreeWorlds[i]);
		else if( i < 2*NUM_TREES/4 )
			drawObject(mTrees[1], mTreeWorlds[i]);
		else if( i < 3*NUM_TREES/4 )
			drawObject(mTrees[2], mTreeWorlds[i]);
		else
			drawObject(mTrees[3], mTreeWorlds[i]);
	}

	HR(gd3dDevice->SetRenderState(D3DRS_ALPHATESTENABLE, false));

	mTerrain->draw();

	//Draw water last due to alpha blending
	mWater->draw();

	HR(mFX->EndPass());
	HR(mFX->End());
```

```
	mGfxStats->display();

	HR(gd3dDevice->EndScene());

	HR(gd3dDevice->Present(0, 0, 0, 0));
}

void RomanCity::buildFX()
{
//Get the FX from the appropriate .fx file.
	ID3DXBuffer* errors = 0;
	HR(D3DXCreateEffectFromFile(gd3dDevice, "NormalMap.fx",
		0, 0, D3DXSHADER_DEBUG, 0, &mFX, &errors));
	if( errors )
		MessageBox(0, (char*)errors->GetBufferPointer(), 0, 0);

	mhTech		= mFX->GetTechniqueByName("NormalMapTech");
	mhWVP		= mFX->GetParameterByName(0, "gWVP");
	mhWorldInv	= mFX->GetParameterByName(0, "gWorldInv");
	mhMtrl		= mFX->GetParameterByName(0, "gMtrl");
	mhLight		= mFX->GetParameterByName(0, "gLight");
	mhEyePosW	= mFX->GetParameterByName(0, "gEyePosW");
	mhWorld		= mFX->GetParameterByName(0, "gWorld");
	mhTex		= mFX->GetParameterByName(0, "gTex");
	mhNormalMap	= mFX->GetParameterByName(0, "gNormalMap");

	HR(mFX->SetMatrix(mhWorldInv, &mSceneWorldInv));
	HR(mFX->SetTechnique(mhTech));
}


void RomanCity::drawObject(Object3D& obj, const D3DXMATRIX& toWorld)
{
	//Use AABB to transform the object, and draw if visible

	AABB box;
	obj.box.xform(toWorld, box);

	if( gCamera->isVisible( box ) )
	{
		HR(mFX->SetMatrix(mhWVP, &(toWorld*gCamera-
>viewProj())));
		D3DXMATRIX worldInvTrans;
		D3DXMatrixInverse(&worldInvTrans, 0, &toWorld);
		D3DXMatrixTranspose(&worldInvTrans, &worldInvTrans);
		HR(mFX->SetMatrix(mhWorldInv, &worldInvTrans));
		HR(mFX->SetMatrix(mhWorld, &toWorld));

		for(UINT j = 0; j < obj.mtrls.size(); ++j)
		{
			HR(mFX->SetValue(mhMtrl, &obj.mtrls[j],
sizeof(Mtrl)));

			if(obj.textures[j] != 0)
			{
				HR(mFX->SetTexture(mhTex, obj.textures[j]));
			}

			else
			{
```

```
                        HR(mFX->SetTexture(mhTex, mWhiteTex));
                  }

                  HR(mFX->CommitChanges());
                  HR(obj.mesh->DrawSubset(j));
            }
      }
      HR(mFX->SetMatrix(mhWorldInv, &mSceneWorldInv));
}

//City generation function
void RomanCity::MakeAnXFile()
{
      //VARIABLE DEFINITION
      BuildCity NewCity;

      int numcount;
      double squaret;
      numcount = 0;

      float td = 5; //city road detail
      float rl = 0.2; //road height
      int citysize = 200; //This is the variable used for adjusting
the entire city circumference.

      //Add the most significant central structures
      NewCity.AddTemple(2,4,1.7+(mTerrain->getHeight(5, 5)));
      NewCity.AddForum(0,-20,2.2+(mTerrain->getHeight(0, -20)));
      NewCity.AddWall(citysize+20,mTerrain->getHeight(0, 0));
      NewCity.AddAmphitheater(100,-104,(mTerrain-
>getHeight(55,64)),14);

      //Add Paving
      for (int i = -citysize; i < citysize; i = i+td){
            for (int j = -citysize; j < citysize; j= j+td){
                  squaret = (i+5)*(i+5)+(j+5)*(j+5);
                  if (sqrt(squaret) < citysize)
                  {
                        //AddPoly(i+td,rl+(mTerrain->getHeight(i+td,
-j)),j,i,rl+(mTerrain->getHeight(i, -j)),j,i+td,rl+(mTerrain-
>getHeight(i+td, -j-td)),j+td,i,rl+(mTerrain->getHeight(i, -j-
td)),j+td,0,1,0);
                  }
            }
      }

      D3DXMATRIX S, T;
      int v;
      v=0;
      float treeScale = GetRandomFloat(0.15f, 0.25f);

      //Add buildings
      for (int i = -citysize; i < citysize; i= i+20){
            for (int j = -citysize; j < citysize; j= j+20){
                  if ((i == 0 && j == 0)||(i == 0 && j == -20)||(i >
60 && j < -80)){
                        //Unavailable land well outside the city
walls. Do nothing
                  } else {
                        squaret = (i+5)*(i+5)+(j+5)*(j+5);
                        if (sqrt(squaret) < (citysize*0.8))
```

```
                        {
                                //Generate the appropriate building
type for each grid square.
                                int buildtype = rand()%7;
                                int tempwi = rand()%4;
                                tempwi = 3;
                                int tempzi = rand()%4;
                                tempzi = 3;
                                float heighest = mTerrain-
>getHeight(i,-j);
                                if (mTerrain->getHeight(i+tempwi+4,-j-
tempzi-4) > heighest) { heighest = mTerrain->getHeight(i+tempwi+4,-j-
tempzi-4);}
                                if (mTerrain->getHeight(i+tempwi+4,-j)
> heighest) { heighest = mTerrain->getHeight(i+tempwi+4,-j);}
                                if (mTerrain->getHeight(i,-j-tempzi-4)
> heighest) { heighest = mTerrain->getHeight(i,-j-tempzi-4);}
                                switch(buildtype)
                                {
                                case 0:
                                        NewCity.AddBuilding(i,j,tempwi +
4,tempzi + 4,rand()%2 + 1, 1.7+heighest,rand()%2);
                                        break;
                                case 1:
                                        NewCity.AddBuilding(i,j,tempwi +
4,tempzi + 4,rand()%2 + 1, 1.7+heighest,rand()%2);
                                        break;
                                case 2:

      NewCity.AddCourtyard(i,j,5,5,1.7+heighest);
                                        treeScale = GetRandomFloat(0.1f,
0.15f);
                                        D3DXMatrixTranslation(&T, (i+5),
(mTerrain->getHeight(i+5,-j-5)), (-j-5));
                                        D3DXMatrixScaling(&S, treeScale,
treeScale, treeScale);
                                        mTreeWorlds[v] = S*T;
                                        v++;

      NewCity.AddBuilding(i+10,j,2,7,1,1.7+heighest+0.02,0);
                                        break;
                                case 3:
                                        NewCity.AddBuilding(i,j,4,7,2,
1.7+heighest,0);

      NewCity.AddBuilding(i+8,j+4,2,3,1, 1.7+heighest,1);
                                        break;
                                case 4:

      NewCity.AddBuilding(i+2,j+4,5,2,1, 1.7+heighest,1);
                                        NewCity.AddBuilding(i+4,j,3,6,1,
1.7+heighest+0.01,0);
                                        break;
                                case 5:

      NewCity.AddBuilding(i+2,j+2,2,4,1, 1.7+heighest,1);

      NewCity.AddBuilding(i+6,j+4,2,5,2, 1.7+heighest+0.01,0);
                                        break;
                                case 6:
```

```
		NewCity.AddCourtyard(i+4,j,5,4,1.7+heighest);
                                        NewCity.AddBuilding(i,j+8,5,2,1,
1.7+heighest+0.01,1);
                                        break;
                                }
                        }
                }
        }

        NewCity.MakeAnX("build2.x");
}

//Road set-up
void RomanCity::buildRoads()
{
        const int widearea = 1000;
        int ** RoadPointArray;

        RoadPointArray = new int*[widearea];
        for(int i = 0; i < widearea; i++)
        {
                RoadPointArray[i] = new int[widearea];
        }

        for(int i = 0; i < widearea; i++)
        {
                for(int j = 0; j < widearea; j++)
                {
                        RoadPointArray[i][j] = 0;
                }
        }

        const int totalroads = 10;
        int secwidth = 10;      //REALLY IMPORTANT VARIABLE
        const float roadheight = 3;
        const float roadwidth = 0.2;

        int currroad = 0;
        //int RoadPointArray[widearea][widearea];
        int RoadJoinArray[totalroads][4];

        BuildCity NewCity;

        int Areacheck = 2;

        double degra = 0.0174532925199433;

        const int MinRoadDist = 2;
        const int MaxRoadDist = 22;
        const int Roadvar = 2;

        int inpx1,inpx2,inpz1,inpz2;
        inpx1 = 0;
        inpz1 = 0;
        inpx2 = 0;
        inpz2 = 0;

        //For every road
        for (int r = 0; r < totalroads; r++){
```

```
		//Assign the start and end points
		if (r==0) {
			inpx1 = 500;
			inpz1 = 500;
		} else {
			do{
				int temppoint = rand()%r;
				inpx1 = RoadJoinArray[temppoint][2];
				inpz1 = RoadJoinArray[temppoint][3];
			} while (RoadPointArray[inpx1][inpz1] > 3);
		}
		int ranang = (rand()%360);
		//OR
		ranang = 90*(rand()%4);
		int ranndist = MinRoadDist+(rand()%Roadvar);
		inpx2 = floor(inpx1+ranndist*sin(degra*ranang));
		inpz2 = floor(inpz1+ranndist*cos(degra*ranang));

		//If the end node is free
			if (RoadPointArray[inpx2][inpz2] < 4){
				//Check all nearby nodes
				for (int i = 0; i < Areacheck; i++){
					for (int j = 0; j < Areacheck; j++){
						if (RoadPointArray[inpx2-
(Areacheck/2)+i][inpz2-(Areacheck/2)+j] > 0){
							if (RoadPointArray[inpx2-
(Areacheck/2)+i][inpz2-(Areacheck/2)+j] > 3){
								//Abort! Too many
roads at this node.
								r = r-1;
							} else {
								//Change end node to
match cross roads
								inpx2 = (inpx2-
(Areacheck/2)+i);
								inpz2 = (inpz2-
(Areacheck/2)+j);
							}
						}
					}
				}
				//Add to the road arrays
				RoadPointArray[inpx1][inpz1] += 1;
				RoadPointArray[inpx2][inpz2] += 1;
				RoadJoinArray[r][0] = inpx1;
				RoadJoinArray[r][1] = inpz1;
				RoadJoinArray[r][2] = inpx2;
				RoadJoinArray[r][3] = inpz2;

				//Now we do the actual point calculation.
				//Move to centre. A little crude.
				inpx1 -=500;
				inpz1-=500;
				inpz2-=500;
				inpx2-=500;

				//Use trigonometry to adjust road position
				if ((inpx2-inpx1) == 0){inpx1-=1;}
				if ((inpz2-inpz1) == 0){inpz1-=1;}
				float gradz = (inpx2-inpx1)/(inpz2-inpz1);
				float gradx = (inpz2-inpz1)/(inpx2-inpx1);
```

```
                        float newang = atan(double ((inpx2-
inpx1)/(inpz2-inpz1)));

                        float tempx1 = inpx1-
(((roadwidth)*cos(newang))-((roadwidth)*sin(newang)));
                        float tempz1 = inpz1-
(((roadwidth)*sin(newang))+((roadwidth)*cos(newang)));
                        float tempx2 =
inpx1+(((roadwidth)*cos(newang))-((roadwidth)*sin(newang)));
                        float tempz2 =
inpz1+(((roadwidth)*sin(newang))+((roadwidth)*cos(newang)));
                        float tempx3 = inpx2-
(((roadwidth)*cos(newang))-((roadwidth)*sin(newang)));
                        float tempz3 = inpz2-
(((roadwidth)*sin(newang))+((roadwidth)*cos(newang)));
                        float tempx4 =
inpx2+(((roadwidth)*cos(newang))-((roadwidth)*sin(newang)));
                        float tempz4 =
inpz2+(((roadwidth)*sin(newang))+((roadwidth)*cos(newang)));


      NewCity.AddPoly(secwidth*tempx1,roadheight+(mTerrain-
>getHeight(-secwidth*tempx1,
secwidth*tempz1)),secwidth*tempz1,secwidth*tempx3,roadheight+(mTerrai
n->getHeight(-
secwidth*tempx3,secwidth*tempz3)),secwidth*tempz3,secwidth*tempx2,roa
dheight+(mTerrain->getHeight(-secwidth*tempx2,
secwidth*tempz2)),secwidth*tempz2,secwidth*tempx4,roadheight+(mTerrai
n->getHeight(-secwidth*tempx4,
secwidth*tempz4)),secwidth*tempz4,0,1,0);

      NewCity.AddPoly(secwidth*tempx1,roadheight+(mTerrain-
>getHeight(-secwidth*tempx1,
secwidth*tempz1)),secwidth*tempz1,secwidth*tempx2,roadheight+(mTerrai
n->getHeight(-
secwidth*tempx2,secwidth*tempz2)),secwidth*tempz2,secwidth*tempx3,roa
dheight+(mTerrain->getHeight(-secwidth*tempx3,
secwidth*tempz3)),secwidth*tempz3,secwidth*tempx4,roadheight+(mTerrai
n->getHeight(-secwidth*tempx4,
secwidth*tempz4)),secwidth*tempz4,0,1,0);
                  } else {
                  //    r = r-1;
                  }
            //}
      }
      NewCity.MakeAnX("build2.x");
}

void RomanCity::buildCity()
{
      //Write an XFile for the whole city. Remove this if the file
has already been created, or if procedural generation is unwanted.
      MakeAnXFile();

      D3DXMATRIX T, Ry;
      ID3DXMesh* tempMesh = 0;

      //We apply the city mesh to a temporary area to calculate the
normals.
```

```
	LoadXFile("build1.x", &mBuildings[0].mesh, mBuildings[0].mtrls,
mBuildings[0].textures);
	LoadXFile("build2.x",&tempMesh, mSceneMtrls, mSceneTextures);

	D3DXMatrixRotationY(&Ry, D3DX_PI);
	D3DXMatrixTranslation(&T, 0.0f, 0.0f, 0.0f);
	mBuildWorlds[0] = Ry*T;
	mBuildWorlds[1] = Ry*T;
	D3DXMatrixRotationY(&Ry, D3DX_PI);
	D3DXMatrixTranslation(&T, 0.0f, 0.0f, 0.0f);
	mRomeWorld = Ry*T;

	//Get vertex decleration adn set up mesh in appropriate format.
	//Clone mesh, apply vertex elements, set to output mesh, then
release.
	D3DVERTEXELEMENT9 elems[MAX_FVF_DECL_SIZE];
	UINT numElems = 0;
	HR(NMapVertex::Decl->GetDeclaration(elems, &numElems));

	ID3DXMesh* clonedTempMesh = 0;
	HR(tempMesh->CloneMesh(D3DXMESH_32BIT | D3DXMESH_MANAGED,
elems, gd3dDevice, &clonedTempMesh));

	HR(D3DXComputeTangentFrameEx(
	  clonedTempMesh,
	  D3DDECLUSAGE_TEXCOORD, 0,
	D3DDECLUSAGE_BINORMAL, 0,
	  D3DDECLUSAGE_TANGENT, 0,
	D3DDECLUSAGE_NORMAL, 0,
	0,
	0,
	  0.01f, 0.25f, 0.01f,
	  &mSceneMesh,
	  0));

	ReleaseCOM(tempMesh);
	ReleaseCOM(clonedTempMesh);

	D3DXMatrixIdentity(&mSceneWorld);
	D3DXMatrixIdentity(&mSceneWorldInv);

	HR(D3DXCreateTextureFromFile(gd3dDevice, "scratches.bmp",
&mSceneNormalMaps[0]));
	HR(D3DXCreateTextureFromFile(gd3dDevice, "newbrick.bmp",
&mSceneNormalMaps[1]));
	HR(D3DXCreateTextureFromFile(gd3dDevice, "col3.bmp",
&mSceneNormalMaps[2]));
	HR(D3DXCreateTextureFromFile(gd3dDevice, "scratches.bmp",
&mSceneNormalMaps[3]));
	HR(D3DXCreateTextureFromFile(gd3dDevice, "whitetex.dds",
&mSceneNormalMaps[4]));
	HR(D3DXCreateTextureFromFile(gd3dDevice, "scratches.bmp",
&mSceneNormalMaps[5]));
	HR(D3DXCreateTextureFromFile(gd3dDevice, "cobblenorm3.jpg",
&mSceneNormalMaps[6]));

	HR(D3DXCreateTextureFromFile(gd3dDevice, "whitetex.dds",
&mWhiteTex));
}

void RomanCity::buildTrees()
```

```
{
	//Loading 4 tree meshes.
	LoadXFile("tree0.x", &mTrees[0].mesh, mTrees[0].mtrls,
mTrees[0].textures);
	LoadXFile("tree1.x", &mTrees[1].mesh, mTrees[1].mtrls,
mTrees[1].textures);
	LoadXFile("tree2.x", &mTrees[2].mesh, mTrees[2].mtrls,
mTrees[2].textures);
	LoadXFile("tree3.x", &mTrees[3].mesh, mTrees[3].mtrls,
mTrees[3].textures);

	for(int i = 0; i < 4; ++i)
	{
		VertexPNT* v = 0;
		HR(mTrees[i].mesh->LockVertexBuffer(0, (void**)&v));
		HR(D3DXComputeBoundingBox(&v->pos, mTrees[i].mesh-
>GetNumVertices(),
			mTrees[i].mesh->GetNumBytesPerVertex(),
			&mTrees[i].box.minPt, &mTrees[i].box.maxPt));
		HR(mTrees[i].mesh->UnlockVertexBuffer());
	}

	//Tree set-up. Transform, scale, and randomized as needed.
	int w = (int)(mTerrain->getWidth() * 0.8f);
	int d = (int)(mTerrain->getDepth() * 0.8f);
	D3DXMATRIX S, T;
	for(int i = 0; i < NUM_TREES; ++i)
	{
		float x = (float)((rand() % w) - (w*0.5f));
		float z = (float)((rand() % d) - (d*0.5f));

		float y = mTerrain->getHeight(x, z) - 0.5f;

		float treeScale = GetRandomFloat(0.15f, 0.25f);

		D3DXMatrixTranslation(&T, x, y, z);
		D3DXMatrixScaling(&S, treeScale, treeScale, treeScale);
		mTreeWorlds[i] = S*T;

		D3DXMatrixTranslation(&T, 0, 40, 0);
		D3DXMatrixScaling(&S, 2, 2, 2);

		//Generate trees within an appropriate height range.
		if(x > -120.0f && x < 120.0f && z > -120.0f && z <
120.0f)
		{
			--i;
		}
	}
}
```

7.2 BUILDCITY.CPP

```
#include "BuildCity.h"
#include "math.h"
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <iostream>
#include <fstream>

using namespace std;

BuildCity::BuildCity(void)
{
	const int rows = 4000000;
	const int cols = 3;

	//Declaration, allocation, and initialization of arrays
	CurrentMat = new int[rows];
	NormDir = new int[rows];

	PointArray = new double*[rows];
	PolyArray = new int*[rows];
	for(int i = 0; i < rows; i++)
	{
		PointArray[i] = new double[cols];
		PolyArray[i] = new int[cols];
	}

	for(int i = 0; i < rows; i++)
	{
		CurrentMat[i] = 0;
		NormDir[i] = 0;
		for(int j = 0; j < cols; j++)
		{
			PointArray[i][j] = 0;
			PolyArray[i][j] = 0;
		}
	}

	//Default values for variables used in generation process
	degra = 0.0174532925199433;
	Secwidth = 2;
	winheight = 1.3;
	winwidth = 0.6;
	floorheight = 4;
	numcount = 0;
	srand(time(0));
	Pointcount = 0;
	Polycount = 0;
	CurrTex = 0;
	numcount = 0;
}

void BuildCity::DeallocateAll(void)
{
	for(int i=0; i<4000000; i++)
	  delete [] PointArray[i];
	delete[] PointArray;
	for(int i=0; i<4000000; i++)
```

```
        delete [] PolyArray[i];
      delete[] PolyArray;
      delete [] CurrentMat;
    delete [] NormDir;
}

BuildCity::~BuildCity(void)
{
}

//Function used for the purpose of rotating a set of points around an
axis a certain number of degrees.
void BuildCity::RotatePoints(double& x1, double& z1, double centerx,
double centerz, double ang){
      float z2,x2;
      x2 = ((x1-centerx)*cos(degra*ang))-((z1-
centerz)*sin(degra*ang))+centerx;
      z2 = ((x1-centerx)*sin(degra*ang))+((z1-
centerz)*cos(degra*ang))+centerz;
      x1 = x2;
      z1 = z2;
}

//Add a polygon to the mesh. This is the lowest level in the "tree"
of generation.
void BuildCity::AddPoly(double x1, double y1, double z1, double x2,
double y2, double z2, double x3, double y3, double z3, double x4,
double y4, double z4, double normx, double normy, double normz){
      PointArray[Pointcount][0] = x2;
      PointArray[Pointcount][2] = y2;
      PointArray[Pointcount][1] = z2;
      Pointcount++;
      PointArray[Pointcount][0] = x1;
      PointArray[Pointcount][2] = y1;
      PointArray[Pointcount][1] = z1;
      Pointcount++;
      PointArray[Pointcount][0] = x3;
      PointArray[Pointcount][2] = y3;
      PointArray[Pointcount][1] = z3;
      Pointcount++;
      PointArray[Pointcount][0] = x4;
      PointArray[Pointcount][2] = y4;
      PointArray[Pointcount][1] = z4;
      Pointcount++;
      if (normx == 1){
            NormDir[Polycount] = 0;
      }
      if (normx == -1){
            NormDir[Polycount] = 1;
      }
      if (normy == 1){
            NormDir[Polycount] = 2;
      }
      if (normy == -1){
            NormDir[Polycount] = 3;
      }
      if (normz == 1){
            NormDir[Polycount] = 4;
      }
      if (normz == -1){
            NormDir[Polycount] = 5;
```

```
	}
	if (normy == 0.5){
		if (normz == 0.5){
			NormDir[Polycount] = 6;
		} else {
			NormDir[Polycount] = 7;
		}
	}
	if (normx == 0.5){
		if (normz == 0.5){
			NormDir[Polycount] = 8;
		} else {
			NormDir[Polycount] = 9;
		}
	}
	if (normx == -0.5){
		if (normz == 0.5){
			NormDir[Polycount] = 10;
		} else {
			NormDir[Polycount] = 11;
		}
	}
	CurrentMat[Polycount] = CurrTex;
	Polycount++;
}

//Define a basic 4-sided object
void BuildCity::AddBox(double x1, double y1, double z1, double x2,
double y2, double z2){
	CurrTex = 1;
	AddPoly(x1,y2,z1, x2,y2,z1, x1,y1,z1, x2,y1,z1, 0,0,-1);
	AddPoly(x1,y1,z2, x2,y1,z2, x1,y2,z2, x2,y2,z2, 0,0,1);
	AddPoly(x1,y2,z2, x1,y2,z1, x1,y1,z2, x1,y1,z1, -1,0,0);
	AddPoly(x2,y2,z1, x2,y2,z2, x2,y1,z1, x2,y1,z2, 1,0,0);
}

//Defining a rotated 4-sided object
void BuildCity::AddRotatedCube(double x1, double y1, double z1,
double x2, double y2, double z2, double ang){
	RotatePoints(x1,z1,x1,z1,ang);
	RotatePoints(x2,z2,x1,z1,ang);
	AddPoly(x1,y2,z1, x2,y2,z1, x1,y1,z1, x2,y1,z1, 0,0,-1);
	AddPoly(x1,y1,z2, x2,y1,z2, x1,y2,z2, x2,y2,z2, 0,0,1);
	AddPoly(x1,y2,z2, x1,y2,z1, x1,y1,z2, x1,y1,z1, -1,0,0);
	AddPoly(x2,y2,z1, x2,y2,z2, x2,y1,z1, x2,y1,z2, 1,0,0);

	AddPoly(x1,y1,z1, x2,y1,z1, x1,y1,z2, x2,y1,z2, 0,-1,0);
	AddPoly(x1,y2,z2, x2,y2,z2, x1,y2,z1, x2,y2,z1, 0,1,0);
}

//Similar to above, but without defaulted textures.
void BuildCity::AddBlankCube(double x1, double y1, double z1, double
x2, double y2, double z2){
	AddPoly(x1,y2,z1, x2,y2,z1, x1,y1,z1, x2,y1,z1, 0,0,-1);
	AddPoly(x1,y1,z2, x2,y1,z2, x1,y2,z2, x2,y2,z2, 0,0,1);
	AddPoly(x1,y2,z2, x1,y2,z1, x1,y1,z2, x1,y1,z1, -1,0,0);
	AddPoly(x2,y2,z1, x2,y2,z2, x2,y1,z1, x2,y1,z2, 1,0,0);

	AddPoly(x1,y1,z1, x2,y1,z1, x1,y1,z2, x2,y1,z2, 0,-1,0);
	AddPoly(x1,y2,z2, x2,y2,z2, x1,y2,z1, x2,y2,z1, 0,1,0);
}
```

```
//As above, but with a top and bottom - a full cuboid shape.
void BuildCity::AddCube(double x1, double y1, double z1, double x2,
double y2, double z2){
	CurrTex = 4;
	AddBox(x1,y1,z1, x2,y2,z2);
	AddPoly(x1,y1,z1, x2,y1,z1, x1,y1,z2, x2,y1,z2, 0,-1,0);
	AddPoly(x1,y2,z2, x2,y2,z2, x1,y2,z1, x2,y2,z1, 0,1,0);
}

//Arch object
void BuildCity::AddArch(double x1, double height, double z1, double
height2, double ang){
	double
x2,z2,x3,z3,x4,z4,x5,z5,x6,z6,x7,z7,x8,z8,x9,z9,x10,z10,x11,z11,x12,z
12,x13,z13,x14,z14,x15,z15,x16,z16;
	float ww;
	float nx,nz;
	if (ang == 90){nx = 0,nz = 1;} else {if (ang == 0) {nx = -1,nz
= 0;} else {if (ang == 0) {nx = 1,nz = 0;} else {nx = 0,nz = -1;}}}
	ww = 1;
	x2 = x1+(Secwidth)/6;   x8 = x1; x9 = x1+(Secwidth)/6;
	x3 = x1+(Secwidth)*2/6; x10 = x1+(Secwidth)*2/6;
	x4 = x1+(Secwidth)*3/6; x11 = x1+(Secwidth)*3/6;
	x5 = x1+(Secwidth)*4/6; x14 = x1+(Secwidth)*4/6;
	x6 = x1+(Secwidth)*5/6; x15 = x1+(Secwidth)*5/6;
	x7 = x1+Secwidth;       x16 = x1+Secwidth;
	z1 = z1;z2 = z1;z3 = z1;z4 = z1;z5 = z1;z6 = z1;z7 = z1;
	z8 = z1-ww;z9 = z1-ww;z10 = z1-ww;z12 = z1-ww;z13 = z1-ww;z14 =
z1-ww;z15 = z1-ww;z16 = z1-ww;z11 = z1-ww;
	RotatePoints(x1,z1,x1,z1,ang); RotatePoints(x8,z8,x1,z1,ang);
	RotatePoints(x2,z2,x1,z1,ang); RotatePoints(x9,z9,x1,z1,ang);
	RotatePoints(x3,z3,x1,z1,ang); RotatePoints(x10,z10,x1,z1,ang);
	RotatePoints(x4,z4,x1,z1,ang); RotatePoints(x11,z11,x1,z1,ang);
	RotatePoints(x5,z5,x1,z1,ang); RotatePoints(x12,z12,x1,z1,ang);
	RotatePoints(x6,z6,x1,z1,ang); RotatePoints(x13,z13,x1,z1,ang);
	RotatePoints(x7,z7,x1,z1,ang); RotatePoints(x14,z14,x1,z1,ang);
	RotatePoints(x15,z15,x1,z1,ang);
RotatePoints(x16,z16,x1,z1,ang);

	AddPoly(  x1,height,z1,x2,height,z2,x1,height2,z1,
x2,height2,z2, nx,0,nz);
	AddPoly( x6,height,z6,x7,height,z7,
x6,height2,z6,x7,height2,z7,nx,0,nz);

	AddPoly(x9,height,z9, x8,height,z8,
x9,height2,z9,x8,height2,z8,  -nx,0,-nz);
	AddPoly(x16,height,z16,x15,height,z15, x16,height2,z16,
x15,height2,z15,   -nx,0,-nz);
	CurrTex = 5;
		//Arch
	AddPoly(x2,height2,z2, x2,height+(height2-height)*4/6,z2,
x3,height2,z3, x3,height+(height2-height)*5/6,z3,  nx,0,nz);
	AddPoly(x3,height2,z3, x3,height+(height2-height)*5/6,z3,
x4,height2,z4, x4,height+(height2-height)*7/8,z4,  nx,0,nz);
	AddPoly(x4,height2,z4, x4,height+(height2-height)*7/8,z4,
x5,height2,z5, x5,height+(height2-height)*5/6,z5,  nx,0,nz);
	AddPoly(x5,height2,z5, x5,height+(height2-height)*5/6,z5,
x6,height2,z6, x6,height+(height2-height)*4/6,z6,  nx,0,nz);
		//Opposite arch
```

```
	AddPoly(x9,height+(height2-height)*4/6,z9,x9,height2,z9,
x10,height+(height2-height)*5/6,z10, x10,height2,z10, -nx,0,-nz);
	AddPoly( x10,height+(height2-height)*5/6,z10,x10,height2,z10,
x11,height+(height2-height)*7/8,z11,  x11,height2,z11, -nx,0,-nz);
	AddPoly( x11,height+(height2-height)*7/8,z11,
x11,height2,z11,x14,height+(height2-height)*5/6,z14,x14,height2,z14,
-nx,0,-nz);
	AddPoly(x14,height+(height2-height)*5/6,z14, x14,height2,z14,
x15,height+(height2-height)*4/6,z15,x15,height2,z15,  -nx,0,-nz);
			//Inner arch
	AddPoly(x2,height+(height2-
height)*4/6,z2,x2,height,z2,x9,height+(height2-
height)*4/6,z9,x9,height,z9,0,-1,0);
	AddPoly(x3,height+(height2-height)*5/6,z3,x2,height+(height2-
height)*4/6,z2,x10,height+(height2-
height)*5/6,z10,x9,height+(height2-height)*4/6,z9,0,-1,0);
	AddPoly(x4,height+(height2-height)*7/8,z4,x3,height+(height2-
height)*5/6,z3,x11,height+(height2-
height)*7/8,z11,x10,height+(height2-height)*5/6,z10,0,-1,0);
	AddPoly(x5,height+(height2-height)*5/6,z5,x4,height+(height2-
height)*7/8,z4,x14,height+(height2-
height)*5/6,z14,x11,height+(height2-height)*7/8,z11,0,-1,0);
	AddPoly(x6,height+(height2-height)*4/6,z6,x5,height+(height2-
height)*5/6,z5,x15,height+(height2-
height)*4/6,z15,x14,height+(height2-height)*5/6,z14,0,-1,0);
	AddPoly(x6,height,z6,x6,height+(height2-
height)*4/6,z6,x15,height,z15,x15,height+(height2-height)*4/6,z15,0,-
1,0);

}

//Roof objects
void BuildCity::AddRoof(double x1, double y1, double z1, double x2,
double y2, double z2) {
	y2+=0.5;
	AddCube(x1-0.4,y1-0.2,z1-0.4,x2+0.4,y1+0.1,z2+0.4);
	AddPoly(x1,y1,z1,(x1+(x2-x1)/2),y2,z1,(x1+(x2-
x1)/2),y1,z1,x2,y1,z1,0,0,-1);
	AddPoly((x1+(x2-x1)/2),y2,z2,x1,y1,z2,x2,y1,z2,(x1+(x2-
x1)/2),y1,z2,0,0,1);
	CurrTex = 1;
	AddPoly((x1+(x2-x1)/2),y2-0.6,z1,x1+0.8,y1,z1,(x1+(x2-
x1)/2),y2,z1-0.2,x1,y1,z1-0.2,0,0,-1);
	AddPoly(x2,y1,z1-0.2,x2-0.8,y1,z1,(x1+(x2-x1)/2),y2,z1-
0.2,(x1+(x2-x1)/2),y2-0.6,z1,0,0,-1);
	AddPoly(x1+0.8,y1,z2,(x1+(x2-x1)/2),y2-
0.6,z2,x1,y1,z2+0.2,(x1+(x2-x1)/2),y2,z2+0.2,0,0,1);
	AddPoly(x2-0.8,y1,z2,x2,y1,z2+0.2,(x1+(x2-x1)/2),y2-
0.6,z2,(x1+(x2-x1)/2),y2,z2+0.2,0,0,1);
	CurrTex = 3;
	AddPoly(x1,y1,z2+0.2, (x1+(x2-x1)/2),y2,z2+0.2, x1,y1,z1-0.2,
(x1+(x2-x1)/2),y2,z1-0.2,  -1,0,0);
	AddPoly((x1+(x2-x1)/2),y2,z2+0.2, x2,y1,z2+0.2, (x1+(x2-
x1)/2),y2,z1-0.2, x2,y1,z1-0.2,  0,0.5,-0.5);
}

void BuildCity::AddRoof2(double x1, double y1, double z1, double x2,
double y2, double z2) {
	y2+=0.5;
	AddCube(x1-0.4,y1-0.2,z1-0.4,x2+0.4,y1+0.1,z2+0.4);
```

```
	AddPoly(x1,y2,(z1+(z2-z1)/2),x1,y1,z1,x1,y1,z2,x1,y1,(z1+(z2-
z1)/2),-1,0,0);
	AddPoly(x2,y1,z1,x2,y2,(z1+(z2-z1)/2),x2,y1,(z1+(z2-
z1)/2),x2,y1,z2,1,0,0);
	CurrTex = 1;
	AddPoly(x1,y1,z1+0.8,x1,y2-0.6,(z1+(z2-z1)/2),x1-0.2,y1,z1,x1-
0.2,y2,(z1+(z2-z1)/2),-1,0,0);
	AddPoly(x1,y1,z2-0.8,x1-0.2,y1,z2,x1,y2-0.6,(z1+(z2-z1)/2),x1-
0.2,y2,(z1+(z2-z1)/2),-1,0,0);
	AddPoly(x2,y2-0.6,(z1+(z2-
z1)/2),x2,y1,z1+0.8,x2+0.2,y2,(z1+(z2-z1)/2),x2+0.2,y1,z1,1,0,0);
	AddPoly(x2+0.2,y1,z2,x2,y1,z2-0.8,x2+0.2,y2,(z1+(z2-
z1)/2),x2,y2-0.6,(z1+(z2-z1)/2),1,0,0);
	CurrTex = 3;
	AddPoly( x2+0.2,y2,(z1+(z2-z1)/2),x2+0.2,y1,z1,  x1-
0.2,y2,(z1+(z2-z1)/2),x1-0.2,y1,z1,  0,0.5,-0.5);
	AddPoly( x2+0.2,y1,z2, x2+0.2,y2,(z1+(z2-z1)/2),x1-0.2,y1,z2,
x1-0.2,y2,(z1+(z2-z1)/2),  -1,0,0);
}

//Basic a hollow cylinder, used for columns
void BuildCity::AddCylinder(double x1, double y1, double z1, double
radius, double radius2, double y2)
{
	AddPoly(x1-radius2,y2,z1+radius2/2, x1-radius2,y2,z1-radius2/2,
x1-radius,y1,z1+radius/2, x1-radius,y1,z1-radius/2, -1,0,0);
	AddPoly(x1+radius2,y2,z1-radius2/2, x1+radius2,y2,z1+radius2/2,
x1+radius,y1,z1-radius/2, x1+radius,y1,z1+radius/2, 1,0,0);
	AddPoly(x1-radius2/2,y2,z1-radius2, x1+radius2/2,y2,z1-radius2,
x1-radius/2,y1,z1-radius, x1+radius/2,y1,z1-radius, 0,0,-1);
	AddPoly( x1+radius2/2,y2,z1+radius2,x1-radius2/2,y2,z1+radius2,
x1+radius/2,y1,z1+radius,x1-radius/2,y1,z1+radius,  0,0,1);
	AddPoly(x1-radius2/2,y2,z1+radius2, x1-radius2,y2,z1+radius2/2,
x1-radius/2,y1,z1+radius, x1-radius,y1,z1+radius/2, -0.5,0,0.5);
	AddPoly( x1+radius2,y2,z1+radius2/2,
x1+radius2/2,y2,z1+radius2,x1+radius,y1,z1+radius/2,x1+radius/2,y1,z1
+radius,   0.5,0,0.5);
	AddPoly( x1-radius2,y2,z1-radius2/2,x1-radius2/2,y2,z1-
radius2,x1-radius,y1,z1-radius/2,x1-radius/2,y1,z1-radius,   -0.5,0,-
0.5);
	AddPoly(x1+radius2/2,y2,z1-radius2, x1+radius2,y2,z1-radius2/2,
x1+radius/2,y1,z1-radius, x1+radius,y1,z1-radius/2, 0.5,0,-0.5);

}

//Basic column
void BuildCity::AddColumn(double x1, double y1, double z1, double
radius, double y2)
{
	radius = 0.3;
	radius = radius+0.2;
	CurrTex = 5;
	AddBlankCube(x1-radius-0.2,y1,z1-radius-
0.2,x1+radius+0.2,y1+0.2,z1+radius+0.2);
	AddBlankCube(x1-radius-0.2,y2-0.2,z1-radius-
0.2,x1+radius+0.2,y2-0,z1+radius+0.2);

	CurrTex = 2;
	AddCylinder(x1,y1,z1,radius,radius-0.15,y2-0.2);
	CurrTex = 5;
	AddCylinder(x1,y1+0.2,z1,radius+0.1,radius+0.2,y1+0.3);
```

```
	AddCylinder(x1,y1+0.3,z1,radius+0.2,radius,y1+0.4);

	AddCylinder(x1,y2-0.4,z1,radius,radius,y2-0.2);
	AddCylinder(x1,y2-0.6,z1,radius-0.2,radius,y2-0.4);
}

//Window objects
void BuildCity::AddWindowx(double x1, double y1, double z1, double
y2){
	CurrTex = 1;
	AddPoly(x1,y1+winheight,z1,  x1+Secwidth,y1+winheight,z1,
x1,y1,z1, x1+Secwidth,y1,z1,0,0,-1);
	AddPoly(x1,y2,z1,  x1+Secwidth,y2,z1, x1,y2-winheight,z1,
x1+Secwidth,y2-winheight,z1,0,0,-1);
	AddPoly(x1,y2-winheight,z1,  x1+winwidth,y2-winheight,z1,
x1,y1+winheight,z1, x1+winwidth,y1+winheight,z1,0,0,-1);
	AddPoly(x1+Secwidth-winwidth,y2-winheight,z1,  x1+Secwidth,y2-
winheight,z1, x1+Secwidth-winwidth,y1+winheight,z1,
x1+Secwidth,y1+winheight,z1,0,0,-1);

	CurrTex = 4;
	AddPoly(x1+winwidth,y2-winheight,z1+(winwidth/2),x1+Secwidth-
winwidth,y2-winheight,z1+(winwidth/2),
x1+winwidth,y1+winheight,z1+(winwidth/2), x1+Secwidth-
winwidth,y1+winheight,z1+(winwidth/2), 0,0,-1);

	CurrTex = 1;
	AddPoly( x1+Secwidth-winwidth,y2-winheight,z1+(winwidth/2),
x1+Secwidth-winwidth,y2-winheight,z1, x1+Secwidth-
winwidth,y1+winheight,z1+(winwidth/2),x1+Secwidth-
winwidth,y1+winheight,z1,-1,0,0);
	AddPoly( x1+winwidth,y1+winheight,z1, x1+winwidth,y2-
winheight,z1,
x1+winwidth,y1+winheight,z1+(winwidth/2),x1+winwidth,y2-
winheight,z1+(winwidth/2),1,0,0);
}

void BuildCity::AddWindowz(double x1, double y1, double z1, double
y2){

	CurrTex = 1;
	AddPoly(x1,y1+winheight,z1,  x1,y1+winheight,z1+Secwidth,
x1,y1,z1, x1,y1,z1+Secwidth,1,0,0);
	AddPoly(x1,y2,z1,  x1,y2,z1+Secwidth, x1,y2-winheight,z1,
x1,y2-winheight,z1+Secwidth,1,0,0);
	AddPoly(x1,y2-winheight,z1,  x1,y2-winheight,z1+winwidth,
x1,y1+winheight,z1, x1,y1+winheight,z1+winwidth,1,0,0);
	AddPoly(x1,y2-winheight,z1+Secwidth-winwidth,  x1,y2-
winheight,z1+Secwidth, x1,y1+winheight,z1+Secwidth-winwidth,
x1,y1+winheight,z1+Secwidth,1,0,0);
	CurrTex = 4;
	AddPoly(x1-(winwidth/2),y2-winheight,z1+winwidth,x1-
(winwidth/2),y2-winheight,z1+Secwidth-winwidth, x1-
(winwidth/2),y1+winheight,z1+winwidth, x1-
(winwidth/2),y1+winheight,z1+Secwidth-winwidth, 1,0,0);
	CurrTex = 1;
	AddPoly( x1-(winwidth/2),y2-winheight,z1+Secwidth-winwidth,
x1,y2-winheight,z1+Secwidth-winwidth, x1-
(winwidth/2),y1+winheight,z1+Secwidth-
winwidth,x1,y1+winheight,z1+Secwidth-winwidth,1,0,0);
```

```
	AddPoly( x1,y1+winheight,z1+winwidth, x1,y2-
winheight,z1+winwidth,   x1-(winwidth/2),y1+winheight,z1+winwidth,x1-
(winwidth/2),y2-winheight,z1+winwidth,-1,0,0);
}

void BuildCity::AddWindowz2(double x1, double y1, double z1, double
y2){
	//AddPoly(x1,y2,z2, x1,y2,z1, x1,y1,z2, x1,y1,z1, -1,0,0);

	CurrTex = 1;
	AddPoly( x1,y1+winheight,z1+Secwidth,
x1,y1+winheight,z1,x1,y1,z1+Secwidth,  x1,y1,z1,-1,0,0);
	AddPoly(x1,y2,z1+Secwidth, x1,y2,z1,x1,y2-
winheight,z1+Secwidth,  x1,y2-winheight,z1, -1,0,0);

	AddPoly(x1,y1+winheight,z1, x1,y1+winheight,z1+winwidth,x1,y2-
winheight,z1,  x1,y2-winheight,z1+winwidth, -1,0,0);
	AddPoly(x1,y1+winheight,z1+Secwidth-winwidth,
x1,y1+winheight,z1+Secwidth,x1,y2-winheight,z1+Secwidth-winwidth,
x1,y2-winheight,z1+Secwidth, -1,0,0);
	CurrTex = 4;
	AddPoly( x1+(winwidth/2),y1+winheight,z1+winwidth,
x1+(winwidth/2),y1+winheight,z1+Secwidth-winwidth,x1+(winwidth/2),y2-
winheight,z1+winwidth,x1+(winwidth/2),y2-winheight,z1+Secwidth-
winwidth, -1,0,0);
	CurrTex = 1;

	AddPoly( x1+(winwidth/2),y2-winheight,z1+winwidth,  x1,y2-
winheight,z1+winwidth,
x1+(winwidth/2),y1+winheight,z1+winwidth,x1,y1+winheight,z1+winwidth,
-1,0,0);
	AddPoly(x1,y2-winheight,z1+Secwidth-winwidth,
x1+(winwidth/2),y2-winheight,z1+Secwidth-
winwidth,x1,y1+winheight,z1+Secwidth-
winwidth,x1+(winwidth/2),y1+winheight,z1+Secwidth-winwidth,1,0,0);

	//AddPoly( x1+winwidth,y1+winheight,z1, x1+winwidth,y2-
winheight,z1,
x1+winwidth,y1+winheight,z1+(winwidth/2),x1+winwidth,y2-
winheight,z1+(winwidth/2),-1,0,0);
}

void BuildCity::AddWindowx2(double x1, double y1, double z1, double
y2){
	CurrTex = 1;
	AddPoly(x1,y1,z1, x1+Secwidth,y1,z1,x1,y1+winheight,z1,
x1+Secwidth,y1+winheight,z1, 0,0,1);
	AddPoly(x1,y2-winheight,z1, x1+Secwidth,y2-
winheight,z1,x1,y2,z1,  x1+Secwidth,y2,z1, 0,0,1);
	AddPoly(x1,y1+winheight,z1, x1+winwidth,y1+winheight,z1,x1,y2-
winheight,z1,  x1+winwidth,y2-winheight,z1, 0,0,1);
	AddPoly(x1+Secwidth-winwidth,y1+winheight,z1,
x1+Secwidth,y1+winheight,z1,x1+Secwidth-winwidth,y2-winheight,z1,
x1+Secwidth,y2-winheight,z1, 0,0,1);

	CurrTex = 4;
	AddPoly(x1+winwidth,y1+winheight,z1-(winwidth/2), x1+Secwidth-
winwidth,y1+winheight,z1-(winwidth/2), x1+winwidth,y2-winheight,z1-
(winwidth/2),x1+Secwidth-winwidth,y2-winheight,z1-(winwidth/2),
0,0,1);
	CurrTex = 1;
```

```
	AddPoly( x1+winwidth,y2-winheight,z1-(winwidth/2),
x1+winwidth,y2-winheight,z1, x1+winwidth,y1+winheight,z1-
(winwidth/2),x1+winwidth,y1+winheight,z1,1,0,0);
	AddPoly( x1+Secwidth-winwidth,y1+winheight,z1,x1+Secwidth-
winwidth,y2-winheight,z1,   x1+Secwidth-winwidth,y1+winheight,z1-
(winwidth/2),x1+Secwidth-winwidth,y2-winheight,z1-(winwidth/2),-
1,0,0);
}

//Forum object
void BuildCity::AddForum(double xvalue, double zvalue, float Yheight)
{
	AddCube(xvalue-0.4,Yheight-5,zvalue-0.4,xvalue+14+0.4,Yheight-
1.5,zvalue+14+0.4);
	AddColumn(xvalue,Yheight-1.5,zvalue,0.5,Yheight+5.5);
	AddColumn(xvalue+4,Yheight-1.5,zvalue,0.5,Yheight+5.5);
	AddColumn(xvalue+10,Yheight-1.5,zvalue,0.5,Yheight+5.5);
	AddColumn(xvalue+14,Yheight-1.5,zvalue,0.5,Yheight+5.5);
	AddColumn(xvalue,Yheight-1.5,zvalue+14,0.5,Yheight+5.5);
	AddColumn(xvalue+4,Yheight-1.5,zvalue+14,0.5,Yheight+5.5);
	AddColumn(xvalue+10,Yheight-1.5,zvalue+14,0.5,Yheight+5.5);
	AddColumn(xvalue+14,Yheight-1.5,zvalue+14,0.5,Yheight+5.5);
	AddColumn(xvalue,Yheight-1.5,zvalue+4,0.5,Yheight+5.5);
	AddColumn(xvalue,Yheight-1.5,zvalue+10,0.5,Yheight+5.5);
	AddColumn(xvalue+14,Yheight-1.5,zvalue+4,0.5,Yheight+5.5);
	AddColumn(xvalue+14,Yheight-1.5,zvalue+10,0.5,Yheight+5.5);
}

//Temple object
void BuildCity::AddTemple(float cx, float cz, float Yheight)
{
	Yheight = Yheight+1;

	//Cella
	CurrTex = 1;
	AddBox(cx+0,Yheight,cz+3.33,cx+1,Yheight+7,cz+13.33);
	AddBox(cx+9,Yheight,cz+3.33,cx+10,Yheight+7,cz+13.33);
	AddBox(cx+1,Yheight,cz+12.33,cx+9,Yheight+7,cz+13.33);
	AddBox(cx+1,Yheight,cz+3.33,cx+4,Yheight+7,cz+4.33);
	AddBox(cx+6,Yheight,cz+3.33,cx+9,Yheight+7,cz+4.33);

	AddBox(cx+4,Yheight+5,cz+3.33,cx+6,Yheight+7,cz+4.33);

	AddCube(cx-0.2,Yheight+6.9,cz-4,cx+10.2,Yheight+8.6,cz+13.33);

	AddColumn(cx+0,Yheight,cz-3.33,0.5,Yheight+7);
	AddColumn(cx+3.33,Yheight,cz+3.33,0.5,Yheight+7);
	AddColumn(cx+10,Yheight,cz-3.33,0.5,Yheight+7);
	AddColumn(cx+6.66,Yheight,cz+3.33,0.5,Yheight+7);
	AddColumn(cx+0,Yheight,cz,0.5,Yheight+7);
	AddColumn(cx+10,Yheight,cz,0.5,Yheight+7);
	AddColumn(cx+0,Yheight,cz+3.33,0.5,Yheight+7);
	AddColumn(cx+0,Yheight,cz+6.66,0.5,Yheight+7);
	AddColumn(cx+0,Yheight,cz+10.0,0.5,Yheight+7);
	AddColumn(cx+0,Yheight,cz+13.33,0.5,Yheight+7);

	AddColumn(cx+3.33,Yheight,cz-3.33,0.5,Yheight+7);
	AddColumn(cx+6.66,Yheight,cz-3.33,0.5,Yheight+7);
	AddColumn(cx+3.33,Yheight,cz+13.33,0.5,Yheight+7);
	AddColumn(cx+6.66,Yheight,cz+13.33,0.5,Yheight+7);
```

```
	AddColumn(cx+10,Yheight,cz+3.33,0.5,Yheight+7);
	AddColumn(cx+10,Yheight,cz+6.66,0.5,Yheight+7);
	AddColumn(cx+10,Yheight,cz+10.0,0.5,Yheight+7);
	AddColumn(cx+10,Yheight,cz+13.33,0.5,Yheight+7);

	//Podium
	CurrTex = 5;
	AddBlankCube(cx-0.4,Yheight-2.5,cz-4,cx+10.4,Yheight-
0.5,cz+14.4);
	AddBlankCube(cx-0.6,Yheight-0.5,cz-4,cx+10.6,Yheight,cz+14.6);
	AddBlankCube(cx-0.4,Yheight-2.5,cz-8,cx+2,Yheight-0.5,cz-4);
	AddBlankCube(cx+8,Yheight-2.5,cz-8,cx+10.4,Yheight-0.5,cz-4);
	AddBlankCube(cx-0.6,Yheight-0.5,cz-8.4,cx+2.2,Yheight,cz-4);
	AddBlankCube(cx+7.8,Yheight-0.5,cz-8.4,cx+10.6,Yheight,cz-4);

	//Stairs
	for (int ii = 0; ii < 9; ii++){
		CurrTex = 5;
	AddBlankCube(cx+2,Yheight-(0.8+(ii*0.2)),cz-
(4.5+(ii*0.4)),cx+8,Yheight-(0.4+(ii*0.2)),cz-(3.5+(ii*0.4)));
	}

	AddRoof(cx-0.3,Yheight+8.6,cz-4,cx+10.3,Yheight+11,cz+13.3);
}
//City wall tower object
void BuildCity::AddTower(double x1, double y1, double z1, double
radius, double y2)
{
	y2 = y2-0.4;
	double radius2 = radius-2.0;
	AddPoly(x1-radius2/2,y2,z1-radius2,x1-radius2,y2,z1-
radius2/2,x1,y2,z1,x1-radius2,y2,z1+radius2/2,0,1,0);
	AddPoly(x1+radius2,y2,z1-radius2/2,x1+radius2/2,y2,z1-
radius2,x1,y2,z1,x1-radius2/2,y2,z1-radius2,0,1,0);
	AddPoly(x1-radius2,y2,z1+radius2/2,x1-
radius2/2,y2,z1+radius2,x1,y2,z1,x1+radius2/2,y2,z1+radius2,0,1,0);
	AddPoly(x1+radius2/2,y2,z1+radius2,x1+radius2,y2,z1+radius2/2,x
1,y2,z1,x1+radius2,y2,z1-radius2/2,0,1,0);

	AddPoly(x1-radius2,y2,z1+radius2/2, x1-radius2,y2,z1-radius2/2,
x1-radius,y1,z1+radius/2, x1-radius,y1,z1-radius/2, -1,0,0);
	AddPoly(x1+radius2,y2,z1-radius2/2, x1+radius2,y2,z1+radius2/2,
x1+radius,y1,z1-radius/2, x1+radius,y1,z1+radius/2, 1,0,0);
	AddPoly(x1-radius2/2,y2,z1-radius2, x1+radius2/2,y2,z1-radius2,
x1-radius/2,y1,z1-radius, x1+radius/2,y1,z1-radius, 0,0,-1);
	AddPoly( x1+radius2/2,y2,z1+radius2,x1-radius2/2,y2,z1+radius2,
x1+radius/2,y1,z1+radius,x1-radius/2,y1,z1+radius,  0,0,1);

	AddPoly(x1-radius2/2,y2,z1+radius2, x1-radius2,y2,z1+radius2/2,
x1-radius/2,y1,z1+radius, x1-radius,y1,z1+radius/2, -0.5,0,0.5);
	AddPoly( x1+radius2,y2,z1+radius2/2,
x1+radius2/2,y2,z1+radius2,x1+radius,y1,z1+radius/2,x1+radius/2,y1,z1
+radius,   0.5,0,0.5);
	AddPoly( x1-radius2,y2,z1-radius2/2,x1-radius2/2,y2,z1-
radius2,x1-radius,y1,z1-radius/2,x1-radius/2,y1,z1-radius,   -0.5,0,-
0.5);
	AddPoly(x1+radius2/2,y2,z1-radius2, x1+radius2,y2,z1-radius2/2,
x1+radius/2,y1,z1-radius, x1+radius,y1,z1-radius/2, 0.5,0,-0.5);
}

//City wall object
```

```
void BuildCity::AddWall(double radius, double height)
{
	double x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4,x5,y5,z5;
	double v = 0;
	float ww = 3;
	double heighttemp, heighttemp2;
	for (int i=0; i<5; i++){
		v = 72*i;
		x1 = radius*sin(degra*v);
		z1 = radius*cos(degra*v);
		v+=72;
		x2 = radius*sin(degra*v);
		z2 = radius*cos(degra*v);
		v = 72*i;
		x3 = (radius+ww)*sin(degra*v);
		z3 = (radius+ww)*cos(degra*v);
		v+=72;
		x4 = (radius+ww)*sin(degra*v);
		z4 = (radius+ww)*cos(degra*v);
		heighttemp = height;
		heighttemp2 = height;
		if (i == 2) {
			heighttemp2 = height-5;
		}
		if (i == 3) {
			heighttemp = height-5;
			heighttemp2 = height-30;
		}
		if (i == 4) {
			heighttemp = height-30;
			heighttemp2 = height-20;
		}
		if (i == 0) {
			heighttemp = height-20;
		}
		CurrTex = 1;
		AddTower(x1,heighttemp,z1,8,heighttemp+25);
		CurrTex = 0;
		AddPoly(x1,heighttemp-10,z1,x1,heighttemp+20,z1,
x2,heighttemp2-10,z2, x2,heighttemp2+20,z2,  -1,0,0);
		AddPoly(x1,heighttemp+20,z1,x3,heighttemp+20,z3,
x2,heighttemp2+20,z2, x4,heighttemp2+20,z4,  0,-1,0);
		AddPoly(x3,heighttemp+20,z3,x3,heighttemp-10,z3,
x4,heighttemp2+20,z4, x4,heighttemp2-10,z4,  -1,0,0);

	}
}

//Amphitheater object
void BuildCity::AddAmphitheater(double centerx, double centerz,
double height, double radius)
{
	double height2 = height+5;
	int sides = 30;
	float ang = 360/sides;
	double x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4,x5,y5,z5;
	double v = 0;
	float ww = 3;
	//Calculate x and z positions for current side
	for (int i=0; i<sides; i++){
		v = ang*i;
```

```
            x1 = radius*sin(degra*v);
            z1 = radius*cos(degra*v);
            v+=ang;
            x2 = radius*sin(degra*v);
            z2 = radius*cos(degra*v);
            v = ang*i;
            x3 = (radius+ww)*sin(degra*v);
            z3 = (radius+ww)*cos(degra*v);
            v+=ang;
            x4 = (radius+ww)*sin(degra*v);
            z4 = (radius+ww)*cos(degra*v);
            AddColumn(centerx+x3, height,
centerz+z3,0.4,height2+0.2);
            AddColumn(centerx+x3, height2,
centerz+z3,0.4,height2+(height2-height)-0.2);
            CurrTex = 5;
            //Top!
            AddPoly(centerx+x1,height2+(height2-
height),centerz+z1,centerx+x3,height2+(height2-height),centerz+z3,
centerx+x2,height2+(height2-height),centerz+z2,
centerx+x4,height2+(height2-height),centerz+z4,  0,-1,0);

            //Repeat for two floors - we do arches here
            for (int j=0; j<2;j++){
                  float diff = (height2-height);
                  //Change the height based on the floor. This is
messy, but it works.
                  if (j==1){
                        height = height2;
                        height2 = height2+diff;
                  }
                  //Facing up

      AddPoly(centerx+x1,height+0.4,centerz+z1,centerx+x3,height+0.4,
centerz+z3, centerx+x2,height+0.4,centerz+z2,
centerx+x4,height+0.4,centerz+z4,  0,1,0);

                  //Outer
                  AddPoly(centerx+(x3+(x4-
x3)/6),height+0.4,centerz+(z3+(z4-z3)/6),centerx+(x3+(x4-
x3)/6),height,centerz+(z3+(z4-z3)/6), centerx+(x3+(x4-
x3)*5/6),height+0.4,centerz+(z3+(z4-z3)*5/6), centerx+(x3+(x4-
x3)*5/6),height,centerz+(z3+(z4-z3)*5/6),  -1,0,0);

      //AddPoly(centerx+x3,height2,centerz+z3,centerx+x3,height,cente
rz+z3, centerx+x4,height2,centerz+z4, centerx+x4,height,centerz+z4,
1,0,0);

      AddPoly(centerx+x3,height2,centerz+z3,centerx+x3,height,centerz
+z3, centerx+(x3+(x4-x3)/6),height2,centerz+(z3+(z4-z3)/6),
centerx+(x3+(x4-x3)/6),height,centerz+(z3+(z4-z3)/6),  -1,0,0);
                  AddPoly(centerx+(x3+(x4-
x3)*5/6),height2,centerz+(z3+(z4-z3)*5/6),centerx+(x3+(x4-
x3)*5/6),height,centerz+(z3+(z4-z3)*5/6),
centerx+x4,height2,centerz+z4, centerx+x4,height,centerz+z4,  -
1,0,0);
                  //Arch
                  AddPoly(centerx+(x3+(x4-
x3)/6),height2,centerz+(z3+(z4-z3)/6), centerx+(x3+(x4-
x3)/6),height+(height2-height)*4/6,centerz+(z3+(z4-z3)/6),
centerx+(x3+(x4-x3)*2/6),height2,centerz+(z3+(z4-z3)*2/6),
```

```
centerx+(x3+(x4-x3)*2/6),height+(height2-height)*5/6,centerz+(z3+(z4-
z3)*2/6),  -1,0,0);
                    AddPoly(centerx+(x3+(x4-
x3)*2/6),height2,centerz+(z3+(z4-z3)*2/6), centerx+(x3+(x4-
x3)*2/6),height+(height2-height)*5/6,centerz+(z3+(z4-z3)*2/6),
centerx+(x3+(x4-x3)*3/6),height2,centerz+(z3+(z4-z3)*3/6),
centerx+(x3+(x4-x3)*3/6),height+(height2-height)*8/9,centerz+(z3+(z4-
z3)*3/6),  -1,0,0);
                    AddPoly(centerx+(x3+(x4-
x3)*3/6),height2,centerz+(z3+(z4-z3)*3/6), centerx+(x3+(x4-
x3)*3/6),height+(height2-height)*8/9,centerz+(z3+(z4-z3)*3/6),
centerx+(x3+(x4-x3)*4/6),height2,centerz+(z3+(z4-z3)*4/6),
centerx+(x3+(x4-x3)*4/6),height+(height2-height)*5/6,centerz+(z3+(z4-
z3)*4/6),  -1,0,0);
                    AddPoly(centerx+(x3+(x4-
x3)*4/6),height2,centerz+(z3+(z4-z3)*4/6), centerx+(x3+(x4-
x3)*4/6),height+(height2-height)*5/6,centerz+(z3+(z4-z3)*4/6),
centerx+(x3+(x4-x3)*5/6),height2,centerz+(z3+(z4-z3)*5/6),
centerx+(x3+(x4-x3)*5/6),height+(height2-height)*4/6,centerz+(z3+(z4-
z3)*5/6),  -1,0,0);
                    //Inner arch
                    AddPoly(centerx+(x3+(x4-x3)/6),height+(height2-
height)*4/6,centerz+(z3+(z4-z3)/6),centerx+(x3+(x4-
x3)/6),height,centerz+(z3+(z4-z3)/6),centerx+(x1+(x2-
x1)/6),height+(height2-height)*4/6,centerz+(z1+(z2-
z1)/6),centerx+(x1+(x2-x1)/6),height,centerz+(z1+(z2-z1)/6),1,0,0);
                    AddPoly(centerx+(x3+(x4-x3)*2/6),height+(height2-
height)*5/6,centerz+(z3+(z4-z3)*2/6),centerx+(x3+(x4-
x3)/6),height+(height2-height)*4/6,centerz+(z3+(z4-
z3)/6),centerx+(x1+(x2-x1)*2/6),height+(height2-
height)*5/6,centerz+(z1+(z2-z1)*2/6),centerx+(x1+(x2-
x1)/6),height+(height2-height)*4/6,centerz+(z1+(z2-z1)/6),1,0,0);
                    AddPoly(centerx+(x3+(x4-x3)*3/6),height+(height2-
height)*8/9,centerz+(z3+(z4-z3)*3/6),centerx+(x3+(x4-
x3)*2/6),height+(height2-height)*5/6,centerz+(z3+(z4-
z3)*2/6),centerx+(x1+(x2-x1)*3/6),height+(height2-
height)*8/9,centerz+(z1+(z2-z1)*3/6),centerx+(x1+(x2-
x1)*2/6),height+(height2-height)*5/6,centerz+(z1+(z2-z1)*2/6),1,0,0);
                    AddPoly(centerx+(x3+(x4-x3)*4/6),height+(height2-
height)*5/6,centerz+(z3+(z4-z3)*4/6),centerx+(x3+(x4-
x3)*3/6),height+(height2-height)*8/9,centerz+(z3+(z4-
z3)*3/6),centerx+(x1+(x2-x1)*4/6),height+(height2-
height)*5/6,centerz+(z1+(z2-z1)*4/6),centerx+(x1+(x2-
x1)*3/6),height+(height2-height)*8/9,centerz+(z1+(z2-z1)*3/6),1,0,0);
                    AddPoly(centerx+(x3+(x4-x3)*5/6),height+(height2-
height)*4/6,centerz+(z3+(z4-z3)*5/6),centerx+(x3+(x4-
x3)*4/6),height+(height2-height)*5/6,centerz+(z3+(z4-
z3)*4/6),centerx+(x1+(x2-x1)*5/6),height+(height2-
height)*4/6,centerz+(z1+(z2-z1)*5/6),centerx+(x1+(x2-
x1)*4/6),height+(height2-height)*5/6,centerz+(z1+(z2-z1)*4/6),1,0,0);
                    AddPoly(centerx+(x3+(x4-
x3)*5/6),height,centerz+(z3+(z4-z3)*5/6),centerx+(x3+(x4-
x3)*5/6),height+(height2-height)*4/6,centerz+(z3+(z4-
z3)*5/6),centerx+(x1+(x2-x1)*5/6),height,centerz+(z1+(z2-
z1)*5/6),centerx+(x1+(x2-x1)*5/6),height+(height2-
height)*4/6,centerz+(z1+(z2-z1)*5/6),1,0,0);

                    if (j==1){
                          height2 = height2-diff;
                          height = height2-diff;
                    }
```

```
            }
            float stairwidth = 0.8;
            float stairheight = 0.3;
            int repeats = 10;
            float stairtop = 5;
            float height3 = height2 + (height2-height);
            //Inner - use for stairs!
            CurrTex = 1;
            AddPoly(centerx+x1,height3-(stairheight)-
stairtop,centerz+z1,centerx+x1,height3,centerz+z1,
centerx+x2,height3-(stairheight)-stairtop,centerz+z2,
centerx+x2,height3,centerz+z2,  -1,0,0);
            CurrTex = 5;
            for (int j=0; j<repeats;j++){
                  v = ang*i;
                  x1 = (radius-(stairwidth*j))*sin(degra*v);
                  z1 = (radius-(stairwidth*j))*cos(degra*v);
                  v+=ang;
                  x2 = (radius-(stairwidth*j))*sin(degra*v);
                  z2 = (radius-(stairwidth*j))*cos(degra*v);
                  v = ang*i;
                  x3 = (radius-(stairwidth*(j+1)))*sin(degra*v);
                  z3 = (radius-(stairwidth*(j+1)))*cos(degra*v);
                  v+=ang;
                  x4 = (radius-(stairwidth*(j+1)))*sin(degra*v);
                  z4 = (radius-(stairwidth*(j+1)))*cos(degra*v);
                  AddPoly(centerx+x3,height3-(stairheight*(j+1))-
stairtop,centerz+z3,centerx+x1,height3-(stairheight*(j+1))-
stairtop,centerz+z1, centerx+x4,height3-(stairheight*(j+1))-
stairtop,centerz+z4, centerx+x2,height3-(stairheight*(j+1))-
stairtop,centerz+z2,  0,-1,0);
                  if (j !=0){
                        AddPoly(centerx+x1,height3-
(stairheight*(j+1))-stairtop,centerz+z1,centerx+x1,height3-
(stairheight*j)-stairtop,centerz+z1, centerx+x2,height3-
(stairheight*(j+1))-stairtop,centerz+z2, centerx+x2,height3-
(stairheight*j)-stairtop,centerz+z2,  -1,0,0);
                  }
                  if (j ==(repeats-1)){

      AddPoly(centerx+x3,height,centerz+z3,centerx+x3,height3-
(stairheight*(j+1))-stairtop,centerz+z3,
centerx+x4,height,centerz+z4, centerx+x4,height3-(stairheight*(j+1))-
stairtop,centerz+z4,  -1,0,0);
                  }
            }
      }
}

//Theater object
void BuildCity::AddTheater(double centerx, double centerz, double
height, double radius)
{
      double height2 = height+5;
      int sides = 30;
      float ang = 360/sides;
      double x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4,x5,y5,z5;
      double v = 0;
      float ww = 3;
      //Calculate x and z positions for current side
      for (int i=0; i<sides/2; i++){
```

```
            v = ang*i;
            x1 = radius*sin(degra*v);
            z1 = radius*cos(degra*v);
            v+=ang;
            x2 = radius*sin(degra*v);
            z2 = radius*cos(degra*v);
            v = ang*i;
            x3 = (radius+ww)*sin(degra*v);
            z3 = (radius+ww)*cos(degra*v);
            v+=ang;
            x4 = (radius+ww)*sin(degra*v);
            z4 = (radius+ww)*cos(degra*v);
            AddColumn(centerx+x3, height,
centerz+z3,0.4,height2+0.2);
            AddColumn(centerx+x3, height2,
centerz+z3,0.4,height2+(height2-height)-0.2);
            CurrTex = 5;
            //Top!
            AddPoly(centerx+x1,height2+(height2-
height),centerz+z1,centerx+x3,height2+(height2-height),centerz+z3,
centerx+x2,height2+(height2-height),centerz+z2,
centerx+x4,height2+(height2-height),centerz+z4,  0,-1,0);

            //Repeat for two floors - we do arches here
            for (int j=0; j<2;j++){
                  float diff = (height2-height);
                  //Change the height based on the floor. This is
messy, but it works.
                  if (j==1){
                        height = height2;
                        height2 = height2+diff;
                  }
                  //Facing up

      AddPoly(centerx+x1,height+0.4,centerz+z1,centerx+x3,height+0.4,
centerz+z3, centerx+x2,height+0.4,centerz+z2,
centerx+x4,height+0.4,centerz+z4,  0,1,0);

                  //Outer
                  AddPoly(centerx+(x3+(x4-
x3)/6),height+0.4,centerz+(z3+(z4-z3)/6),centerx+(x3+(x4-
x3)/6),height,centerz+(z3+(z4-z3)/6), centerx+(x3+(x4-
x3)*5/6),height+0.4,centerz+(z3+(z4-z3)*5/6), centerx+(x3+(x4-
x3)*5/6),height,centerz+(z3+(z4-z3)*5/6),  -1,0,0);

      //AddPoly(centerx+x3,height2,centerz+z3,centerx+x3,height,cente
rz+z3, centerx+x4,height2,centerz+z4, centerx+x4,height,centerz+z4,
1,0,0);

      AddPoly(centerx+x3,height2,centerz+z3,centerx+x3,height,centerz
+z3, centerx+(x3+(x4-x3)/6),height2,centerz+(z3+(z4-z3)/6),
centerx+(x3+(x4-x3)/6),height,centerz+(z3+(z4-z3)/6),  -1,0,0);
                  AddPoly(centerx+(x3+(x4-
x3)*5/6),height2,centerz+(z3+(z4-z3)*5/6),centerx+(x3+(x4-
x3)*5/6),height,centerz+(z3+(z4-z3)*5/6),
centerx+x4,height2,centerz+z4, centerx+x4,height,centerz+z4,  -
1,0,0);
                  //Arch
                  AddPoly(centerx+(x3+(x4-
x3)/6),height2,centerz+(z3+(z4-z3)/6), centerx+(x3+(x4-
x3)/6),height+(height2-height)*4/6,centerz+(z3+(z4-z3)/6),
```

```
centerx+(x3+(x4-x3)*2/6),height2,centerz+(z3+(z4-z3)*2/6),
centerx+(x3+(x4-x3)*2/6),height+(height2-height)*5/6,centerz+(z3+(z4-
z3)*2/6),  -1,0,0);
                    AddPoly(centerx+(x3+(x4-
x3)*2/6),height2,centerz+(z3+(z4-z3)*2/6), centerx+(x3+(x4-
x3)*2/6),height+(height2-height)*5/6,centerz+(z3+(z4-z3)*2/6),
centerx+(x3+(x4-x3)*3/6),height2,centerz+(z3+(z4-z3)*3/6),
centerx+(x3+(x4-x3)*3/6),height+(height2-height)*8/9,centerz+(z3+(z4-
z3)*3/6),  -1,0,0);
                    AddPoly(centerx+(x3+(x4-
x3)*3/6),height2,centerz+(z3+(z4-z3)*3/6), centerx+(x3+(x4-
x3)*3/6),height+(height2-height)*8/9,centerz+(z3+(z4-z3)*3/6),
centerx+(x3+(x4-x3)*4/6),height2,centerz+(z3+(z4-z3)*4/6),
centerx+(x3+(x4-x3)*4/6),height+(height2-height)*5/6,centerz+(z3+(z4-
z3)*4/6),  -1,0,0);
                    AddPoly(centerx+(x3+(x4-
x3)*4/6),height2,centerz+(z3+(z4-z3)*4/6), centerx+(x3+(x4-
x3)*4/6),height+(height2-height)*5/6,centerz+(z3+(z4-z3)*4/6),
centerx+(x3+(x4-x3)*5/6),height2,centerz+(z3+(z4-z3)*5/6),
centerx+(x3+(x4-x3)*5/6),height+(height2-height)*4/6,centerz+(z3+(z4-
z3)*5/6),  -1,0,0);
                    //Inner arch
                    AddPoly(centerx+(x3+(x4-x3)/6),height+(height2-
height)*4/6,centerz+(z3+(z4-z3)/6),centerx+(x3+(x4-
x3)/6),height,centerz+(z3+(z4-z3)/6),centerx+(x1+(x2-
x1)/6),height+(height2-height)*4/6,centerz+(z1+(z2-
z1)/6),centerx+(x1+(x2-x1)/6),height,centerz+(z1+(z2-z1)/6),1,0,0);
                    AddPoly(centerx+(x3+(x4-x3)*2/6),height+(height2-
height)*5/6,centerz+(z3+(z4-z3)*2/6),centerx+(x3+(x4-
x3)/6),height+(height2-height)*4/6,centerz+(z3+(z4-
z3)/6),centerx+(x1+(x2-x1)*2/6),height+(height2-
height)*5/6,centerz+(z1+(z2-z1)*2/6),centerx+(x1+(x2-
x1)/6),height+(height2-height)*4/6,centerz+(z1+(z2-z1)/6),1,0,0);
                    AddPoly(centerx+(x3+(x4-x3)*3/6),height+(height2-
height)*8/9,centerz+(z3+(z4-z3)*3/6),centerx+(x3+(x4-
x3)*2/6),height+(height2-height)*5/6,centerz+(z3+(z4-
z3)*2/6),centerx+(x1+(x2-x1)*3/6),height+(height2-
height)*8/9,centerz+(z1+(z2-z1)*3/6),centerx+(x1+(x2-
x1)*2/6),height+(height2-height)*5/6,centerz+(z1+(z2-z1)*2/6),1,0,0);
                    AddPoly(centerx+(x3+(x4-x3)*4/6),height+(height2-
height)*5/6,centerz+(z3+(z4-z3)*4/6),centerx+(x3+(x4-
x3)*3/6),height+(height2-height)*8/9,centerz+(z3+(z4-
z3)*3/6),centerx+(x1+(x2-x1)*4/6),height+(height2-
height)*5/6,centerz+(z1+(z2-z1)*4/6),centerx+(x1+(x2-
x1)*3/6),height+(height2-height)*8/9,centerz+(z1+(z2-z1)*3/6),1,0,0);
                    AddPoly(centerx+(x3+(x4-x3)*5/6),height+(height2-
height)*4/6,centerz+(z3+(z4-z3)*5/6),centerx+(x3+(x4-
x3)*4/6),height+(height2-height)*5/6,centerz+(z3+(z4-
z3)*4/6),centerx+(x1+(x2-x1)*5/6),height+(height2-
height)*4/6,centerz+(z1+(z2-z1)*5/6),centerx+(x1+(x2-
x1)*4/6),height+(height2-height)*5/6,centerz+(z1+(z2-z1)*4/6),1,0,0);
                    AddPoly(centerx+(x3+(x4-
x3)*5/6),height,centerz+(z3+(z4-z3)*5/6),centerx+(x3+(x4-
x3)*5/6),height+(height2-height)*4/6,centerz+(z3+(z4-
z3)*5/6),centerx+(x1+(x2-x1)*5/6),height,centerz+(z1+(z2-
z1)*5/6),centerx+(x1+(x2-x1)*5/6),height+(height2-
height)*4/6,centerz+(z1+(z2-z1)*5/6),1,0,0);

                    if (j==1){
                          height2 = height2-diff;
                          height = height2-diff;
```

```
                }
            }
            float stairwidth = 0.8;
            float stairheight = 0.3;
            int repeats = 10;
            float stairtop = 5;
            float height3 = height2 + (height2-height);
            //Inner - use for stairs!
            CurrTex = 1;
            AddPoly(centerx+x1,height3-(stairheight)-
stairtop,centerz+z1,centerx+x1,height3,centerz+z1,
centerx+x2,height3-(stairheight)-stairtop,centerz+z2,
centerx+x2,height3,centerz+z2,  -1,0,0);
            CurrTex = 5;
            for (int j=0; j<repeats;j++){
                  v = ang*i;
                  x1 = (radius-(stairwidth*j))*sin(degra*v);
                  z1 = (radius-(stairwidth*j))*cos(degra*v);
                  v+=ang;
                  x2 = (radius-(stairwidth*j))*sin(degra*v);
                  z2 = (radius-(stairwidth*j))*cos(degra*v);
                  v = ang*i;
                  x3 = (radius-(stairwidth*(j+1)))*sin(degra*v);
                  z3 = (radius-(stairwidth*(j+1)))*cos(degra*v);
                  v+=ang;
                  x4 = (radius-(stairwidth*(j+1)))*sin(degra*v);
                  z4 = (radius-(stairwidth*(j+1)))*cos(degra*v);
                  AddPoly(centerx+x3,height3-(stairheight*(j+1))-
stairtop,centerz+z3,centerx+x1,height3-(stairheight*(j+1))-
stairtop,centerz+z1, centerx+x4,height3-(stairheight*(j+1))-
stairtop,centerz+z4, centerx+x2,height3-(stairheight*(j+1))-
stairtop,centerz+z2,  0,-1,0);
                  if (j !=0){
                        AddPoly(centerx+x1,height3-
(stairheight*(j+1))-stairtop,centerz+z1,centerx+x1,height3-
(stairheight*j)-stairtop,centerz+z1, centerx+x2,height3-
(stairheight*(j+1))-stairtop,centerz+z2, centerx+x2,height3-
(stairheight*j)-stairtop,centerz+z2,  -1,0,0);
                  }
                  if (j ==(repeats-1)){

      AddPoly(centerx+x3,height,centerz+z3,centerx+x3,height3-
(stairheight*(j+1))-stairtop,centerz+z3,
centerx+x4,height,centerz+z4, centerx+x4,height3-(stairheight*(j+1))-
stairtop,centerz+z4,  -1,0,0);
                  }
            }
      }
}

//Courtyard, for use in villa objects
void BuildCity::AddCourtyard(double centerx, double centerz, int
xrows, int zrows, float landy)
{
      CurrTex = 1;
      AddBlankCube(centerx-0.2,-8+landy,centerz-
0.2,centerx+(xrows*Secwidth)+0.2,-
1+landy,centerz+(zrows*Secwidth)+0.2);
      AddRoof2(centerx,-
1.5+landy+floorheight,centerz,centerx+(Secwidth*xrows),landy+floorhei
ght-1,centerz+(Secwidth));
```

```
	AddRoof2(centerx,-
1.5+landy+floorheight,centerz+(Secwidth*zrows)-
Secwidth,centerx+(Secwidth*xrows),landy+floorheight-
1,centerz+(Secwidth*zrows));
	AddRoof(centerx+0.01,-
1.5+landy+floorheight+0.01,centerz+Secwidth,centerx+Secwidth+0.01,lan
dy+floorheight-1,centerz+(Secwidth*zrows)-Secwidth);
	AddRoof(centerx+(Secwidth*xrows)-Secwidth+0.01,-
1.5+landy+floorheight+0.01,centerz+Secwidth,centerx+(Secwidth*xrows)+
0.01,landy+floorheight-1,centerz+(Secwidth*zrows)-Secwidth);
	CurrTex = 5;
	for (int i = 1; i < xrows+1; i++){
		AddArch(centerx+(Secwidth*i),-1.5+landy,centerz,-
1.5+landy+floorheight,180);
	}
	for (int i = 0; i < xrows; i++){
		AddArch(centerx+(Secwidth*i),-
1.5+landy,centerz+(Secwidth*zrows),-1.5+landy+floorheight,0);
	}
	for (int i = 0; i < zrows; i++){
		AddArch(centerx,-1.5+landy,centerz+(Secwidth*i),-
1.5+landy+floorheight,90);
	}
	for (int i = 1; i < zrows+1; i++){
		AddArch(centerx+(Secwidth*xrows),-
1.5+landy,centerz+(Secwidth*i),-1.5+landy+floorheight,270);
	}
}

//Generic, governmental insula
void BuildCity::AddBuilding(double centerx, double centerz, int
xrows, int zrows, int hrows, float landy, int rooftype)
{
	CurrTex = 5;
	AddBlankCube(centerx-0.2,-8+landy,centerz-
0.2,centerx+(xrows*Secwidth)+0.2,-
1+landy,centerz+(zrows*Secwidth)+0.2);
	for (int i = 0; i < xrows; i++){
		for (int j = 0; j < hrows; j++){
			if (j==0){
				int rano = rand()%6;
				//CurrTex = 1;
				if (rano == 0) {
					CurrTex = 2;
					AddArch(centerx+(xrows*Secwidth)-
(Secwidth*i)-Secwidth,floorheight*j-
1.5+landy,centerz+(zrows*Secwidth),floorheight*j+floorheight-
1.5+landy,0);
					CurrTex = 1;
					AddBlankCube(centerx+(xrows*Secwidth)-
(Secwidth*i)-Secwidth,floorheight*j-
4.0+landy,centerz+(zrows*Secwidth)-Secwidth,centerx+(xrows*Secwidth)-
(Secwidth*i),floorheight*j-
1.5+landy,centerz+(zrows*Secwidth)+(Secwidth*0.3));

	AddBlankCube(centerx+(xrows*Secwidth)+0.01-(Secwidth*i)-
Secwidth,floorheight*j-4.0+landy,centerz+0.01+(zrows*Secwidth)-
Secwidth,centerx+(xrows*Secwidth)+0.01-(Secwidth*i),floorheight*j-
1.9+landy,centerz+0.01+(zrows*Secwidth)+(Secwidth*0.6));
					AddBlankCube(centerx+(xrows*Secwidth)-
0.01-(Secwidth*i)-Secwidth,floorheight*j-4.0+landy,centerz-
```

```
0.01+(zrows*Secwidth)-Secwidth,centerx+(xrows*Secwidth)-0.01-
(Secwidth*i),floorheight*j-2.3+landy,centerz-
0.01+(zrows*Secwidth)+(Secwidth*0.9));
                              AddBlankCube(centerx+(xrows*Secwidth)-
0.02-(Secwidth*i)-Secwidth,floorheight*j-4.0+landy,centerz-
0.02+(zrows*Secwidth)-Secwidth,centerx+(xrows*Secwidth)-0.02-
(Secwidth*i),floorheight*j-2.7+landy,centerz-
0.02+(zrows*Secwidth)+(Secwidth*1.2));

      AddWindowx(centerx+(Secwidth*i),floorheight*j-
1.5+landy,centerz,floorheight*j+floorheight-1.5+landy);
                        } else {

      AddWindowx(centerx+(Secwidth*i),floorheight*j-
1.5+landy,centerz,floorheight*j+floorheight-1.5+landy);
                              AddWindowx2(centerx+(xrows*Secwidth)-
(Secwidth*i)-Secwidth,floorheight*j-
1.5+landy,centerz+(zrows*Secwidth),floorheight*j+floorheight-
1.5+landy);
                        }

                  } else {
                  AddWindowx(centerx+(Secwidth*i),floorheight*j-
1.5+landy,centerz,floorheight*j+floorheight-1.5+landy);
                  AddWindowx2(centerx+(xrows*Secwidth)-(Secwidth*i)-
Secwidth,floorheight*j-
1.5+landy,centerz+(zrows*Secwidth),floorheight*j+floorheight-
1.5+landy);
                  }
            }
      }
      for (int i = 0; i < zrows; i++){
            for (int j = 0; j < hrows; j++){
                  AddWindowz(centerx+(xrows*Secwidth),floorheight*j-
1.5+landy,centerz+(Secwidth*i),floorheight*j+floorheight-1.5+landy);
                  AddWindowz2(centerx,floorheight*j-
1.5+landy,centerz+(Secwidth*i),floorheight*j+floorheight-1.5+landy);
            }
      }
      if (rooftype == 1){
            AddRoof2(centerx,floorheight*hrows-
1.5+landy,centerz,centerx+(xrows*Secwidth),floorheight*hrows+0.5+land
y,centerz+(zrows*Secwidth));
      } else {
            AddRoof(centerx,floorheight*hrows-
1.5+landy,centerz,centerx+(xrows*Secwidth),floorheight*hrows+0.5+land
y,centerz+(zrows*Secwidth));
      }
}

double BuildCity::ReturnPoint(int p1, int p2)
{
      return PointArray[p1][p2];
}
int BuildCity::ReturnPoly(int p1, int p2)
{
      return PolyArray[p1][p2];
}
int BuildCity::ReturnCurrMat(int p1)
{
      return CurrentMat[p1];
```

```
}
int BuildCity::ReturnNorm(int p1)
{
	return NormDir[p1];
}
int BuildCity::ReturnPointCount(void)
{
	return Pointcount;
}
int BuildCity::ReturnPolyCount(void)
{
	return Polycount;
}

//Once the city has been generated and stored in the arrays, stream
the data into a .x file
void BuildCity::MakeAnX(string filename)
{
	//Set up input/output files
	//SetCurrentDirectory("Art/");
	string line;
	ofstream myfile;
	myfile.open (filename);
	ifstream opfile ("XFilePart1.txt");

	//Output the contents of example.txt
	if (opfile.is_open())
	{
		while ( opfile.good() )
		{
			getline (opfile,line);
			myfile << line << endl;
		}
		opfile.close();
	}

	//Output Pointcount
	 myfile << Pointcount << ";" << endl;

	for (int i = 0; i<Pointcount;i++)
	{
		myfile << fixed << PointArray[i][0] <<  ";" <<
PointArray[i][1] << ";" << PointArray[i][2] << ";";

		myfile << endl;
	}
	myfile << Polycount << ";"<< endl;
	for (int i = 0; i < Polycount; i++)
	{
		myfile << "4;" << numcount << ",";
		numcount++;
		myfile << numcount << ",";
		numcount++;
		myfile << numcount << ",";
		numcount++;
		myfile << numcount << ";," << endl;
		numcount++;
	}
	myfile << "MeshMaterialList {" << endl;
	myfile << "7;" << endl; //NUMBER OF MATERIALS
	myfile << Polycount << ";" << endl;
```

```
	for (int i = 0; i < Polycount; i++)
	{
		myfile << CurrentMat[i];
		if (i == Polycount-1) {
			myfile << ";;" << endl;
		} else {
			myfile << "," << endl;
		}
	}

	ifstream opfile2 ("XFilePart2.txt");
	if (opfile2.is_open())
	{
		while ( opfile2.good() )
		{
			getline (opfile2,line);
			myfile << line << endl;
		}
		opfile2.close();
	}

	myfile << Polycount << ";" << endl;
	for (int i = 0; i < Polycount; i++)
	{
		myfile << "4;" << NormDir[i] << "," << NormDir[i] << ","
<< NormDir[i] << "," << NormDir[i] << ";";

		if (i == Polycount-1){
			myfile << ";" << endl;
		} else {
			myfile << "," << endl;
		}
		numcount++;

	}

	myfile << "}" << endl;
	myfile << "MeshTextureCoords {" << endl;
	myfile << Pointcount << ";" << endl;
	for (int i= 0;  i< Pointcount; i=i+4){
		myfile << "0.0000;0.0000;" << endl;
		myfile << "0.0000;1.0000;" << endl;
		myfile << "1.0000;1.0000;" << endl;
		myfile << "1.0000;0.0000;";
		if (i == Pointcount-1){
				myfile << ";" << endl;
		} else {
				myfile << endl;
		}
	}
	myfile << "}" << endl << "}" << endl << "}";
	myfile.close();

	DeallocateAll();
}
```

7.3 XFILEPART1.TXT

```
xof 0302txt 0032
Header {
 1;
 0;
 1;
}
template Header {
 <3D82AB43-62DA-11cf-AB39-0020AF71E433>
 WORD major;
 WORD minor;
 DWORD flags;
}

template Vector {
 <3D82AB5E-62DA-11cf-AB39-0020AF71E433>
 FLOAT x;
 FLOAT y;
 FLOAT z;
}

template Coords2d {
 <F6F23F44-7686-11cf-8F52-0040333594A3>
 FLOAT u;
 FLOAT v;
}

template Matrix4x4 {
 <F6F23F45-7686-11cf-8F52-0040333594A3>
 array FLOAT matrix[16];
}

template ColorRGBA {
 <35FF44E0-6C7C-11cf-8F52-0040333594A3>
 FLOAT red;
 FLOAT green;
 FLOAT blue;
 FLOAT alpha;
}

template ColorRGB {
 <D3E16E81-7835-11cf-8F52-0040333594A3>
 FLOAT red;
 FLOAT green;
 FLOAT blue;
}

template TextureFilename {
 <A42790E1-7810-11cf-8F52-0040333594A3>
 STRING filename;
}

template Material {
 <3D82AB4D-62DA-11cf-AB39-0020AF71E433>
 ColorRGBA faceColor;
 FLOAT power;
 ColorRGB specularColor;
 ColorRGB emissiveColor;
```

```
 [...]
}

template MeshFace {
 <3D82AB5F-62DA-11cf-AB39-0020AF71E433>
 DWORD nFaceVertexIndices;
 array DWORD faceVertexIndices[nFaceVertexIndices];
}

template MeshTextureCoords {
 <F6F23F40-7686-11cf-8F52-0040333594A3>
 DWORD nTextureCoords;
 array Coords2d textureCoords[nTextureCoords];
}

template MeshMaterialList {
 <F6F23F42-7686-11cf-8F52-0040333594A3>
 DWORD nMaterials;
 DWORD nFaceIndexes;
 array DWORD faceIndexes[nFaceIndexes];
 [Material]
}

template MeshNormals {
 <F6F23F43-7686-11cf-8F52-0040333594A3>
 DWORD nNormals;
 array Vector normals[nNormals];
 DWORD nFaceNormals;
 array MeshFace faceNormals[nFaceNormals];
}

template Mesh {
 <3D82AB44-62DA-11cf-AB39-0020AF71E433>
 DWORD nVertices;
 array Vector vertices[nVertices];
 DWORD nFaces;
 array MeshFace faces[nFaces];
 [...]
}

template FrameTransformMatrix {
 <F6F23F41-7686-11cf-8F52-0040333594A3>
 Matrix4x4 frameMatrix;
}

template Frame {
 <3D82AB46-62DA-11cf-AB39-0020AF71E433>
 [...]
}
template FloatKeys {
 <10DD46A9-775B-11cf-8F52-0040333594A3>
 DWORD nValues;
 array FLOAT values[nValues];
}

template TimedFloatKeys {
 <F406B180-7B3B-11cf-8F52-0040333594A3>
 DWORD time;
 FloatKeys tfkeys;
}
```

```
template AnimationKey {
 <10DD46A8-775B-11cf-8F52-0040333594A3>
 DWORD keyType;
 DWORD nKeys;
 array TimedFloatKeys keys[nKeys];
}

template AnimationOptions {
 <E2BF56C0-840F-11cf-8F52-0040333594A3>
 DWORD openclosed;
 DWORD positionquality;
}

template Animation {
 <3D82AB4F-62DA-11cf-AB39-0020AF71E433>
 [...]
}

template AnimationSet {
 <3D82AB50-62DA-11cf-AB39-0020AF71E433>
 [Animation]
}

template XSkinMeshHeader {
 <3cf169ce-ff7c-44ab-93c0-f78f62d172e2>
 WORD nMaxSkinWeightsPerVertex;
 WORD nMaxSkinWeightsPerFace;
 WORD nBones;
}

template VertexDuplicationIndices {
 <b8d65549-d7c9-4995-89cf-53a9a8b031e3>
 DWORD nIndices;
 DWORD nOriginalVertices;
 array DWORD indices[nIndices];
}

template SkinWeights {
 <6f0d123b-bad2-4167-a0d0-80224f25fabb>
 STRING transformNodeName;
 DWORD nWeights;
 array DWORD vertexIndices[nWeights];
 array FLOAT weights[nWeights];
 Matrix4x4 matrixOffset;
}

Frame polySurface12 {
   FrameTransformMatrix {
1.000000,0.000000,0.000000,0.000000,
0.000000,-0.000000,-1.000000,0.000000,
0.000000,1.000000,-0.000000,0.000000,
0.000000,0.000000,0.000000,1.000000;;
 }
Mesh polySurface121 {
```

7.4 XFILEPART2.TXT

```
//City Wall
Material {
 1.000000;1.000000;1.000000;1.000000;;
25.000000;
 1.000000;1.000000;1.000000;;
 0.200000;0.200000;0.200000;;
TextureFilename {
"newwall.png";
}
 }
 //Bricks
Material {
 1.000000;1.000000;1.000000;1.000000;;
25.000000;
 1.000000;1.000000;1.000000;;
 0.200000;0.200000;0.200000;;
TextureFilename {
"stone.dds";
}
 }
 //Columns
Material {
 1.000000;1.000000;1.000000;1.000000;;
25.000000;
 1.000000;1.000000;1.000000;;
 0.200000;0.200000;0.200000;;
TextureFilename {
"stone.dds";
}
 }
 //Roof
Material {
 1.000000;1.000000;1.000000;1.000000;;
25.000000;
 1.000000;1.000000;1.000000;;
 0.200000;0.200000;0.200000;;
TextureFilename {
"roof2.jpg";
}
 }
 //Window
Material {
 1.000000;1.000000;1.000000;0.500000;;
20.000000;
 1.000000;1.000000;1.000000;;
 0.200000;0.200000;0.200000;;
TextureFilename {
"window1.bmp";
}
 }
  //Granite
Material {
 1.000000;1.000000;1.000000;1.000000;;
2.000000;
 1.000000;1.000000;1.000000;;
 0.200000;0.200000;0.200000;;
TextureFilename {
```

```
"stone.dds";
}
 }
  //Floor
Material {
 1.000000;1.000000;1.000000;1.000000;;
2.000000;
 1.000000;1.000000;1.000000;;
 0.200000;0.200000;0.200000;;
TextureFilename {
"stone.dds";
}
 }
}

 MeshNormals {
 12;
1.00000;0.000000;0.000000;,
-1.000000;0.000000;0.000000;,
0.000000;0.000000;1.000000;,
0.000000;0.000000;-1.000000;,
0.000000;1.000000;0.000000;,
0.000000;-1.000000;0.000000;,
0.000000;0.500000;0.500000;,
0.000000;-0.500000;0.500000;,
0.500000;0.500000;0.000000;,
0.500000;-0.500000;0.000000;,
-0.500000;0.500000;0.000000;,
-0.500000;-0.500000;0.000000;;
```